

Final Year Project

Keyword Spotting on an Edge Device Using Raw Audio

Cian Ferriter

Student ID: 17392773

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Professor Guénolé Silvestre



UCD School of Computer Science

University College Dublin

May 7, 2021

Table of Contents

1	Introduction	5
2	Related Work and Ideas	7
2.1	Neural Networks for KWS	7
2.2	Energy Efficiency	9
2.3	Feature Extraction and Filterbanks	9
2.4	Dataset: Google Speech Commands	10
3	Project Workplan	13
3.1	Accuracy	13
3.2	Timeline	13
4	Summary and Conclusions	15
5	Core Contributions	16
5.1	Model Architectures/Breakdown	17
5.2	HyperParameter/Model Tuning	20
6	Software Engineering/ Deployment	26
6.1	Edge Device as a Security Assistant	28
6.2	Approach	28
7	Evaluation	30
8	Future Work	36
9	Conclusion	37

Details:

- Gitlab [HeyIntruder! Keyword Spotting on an Edge Device Using Raw Audio](#)

Report Notes:

The following are additional sections added as part of the final report:

- Summary & Conclusions
- Core Contributions
- Software Engineering / Deployment
- Evaluation
- Future Work
- Conclusion

The following adjustments were made to the interim report:

- Addition of section on training deep models (Related Work)
- Update on Baseline model from being an HMM-DNN hybrid to being an MFCC trained CNN

List of Common Abbreviations Used Within

ANN: Artificial Neural Network

MFCC: Mel Frequency Cepstrum Co-Efficients

CNN: Convolutional Neural Network

DSP: Digital Signal Processing

KWS: KeyWord Spotting

IoT: Internet of Things

HCI: Human Computer Interaction

DL: Deep Learning

ODL: Online Deep Learning

DSConv: Depthwise Separable Convolutions

GSP: Google Speech Commands

Abstract

This project endeavours to build a portable KeyWord Spotting (KWS) neural network which can be used to accurately classify certain keywords from a spoken stream of raw audio data on an edge device.

To accomplish this a number of different neural network architectures are implemented, trained, and evaluated before being ported to a raspberry-pi device for general real-life use. This paper identifies state of the art architectures which are used in the final neural network model. A number of comparisons have been made between the different architectures to identify the most suitable architecture for completion. The main focus being placed on a direct evaluation of two factors – A computationally efficient model (low parameters) vs high classification accuracy. The chosen architecture is compared against a model based on the use of Mel Frequency Cepstrum Co-efficients, a data pre-processing technique which has been the backbone of KWS systems since their inception. The various models are trained on Google's Speech Commands dataset [1].

The project achieves a best accuracy of 94% using state of the art techniques and modelling. This model is successfully deployed to an edge device which engages with live inferences and task completion.

Project Specification

Project Name:

'Hey Intruder!'

This project will develop a keyword spotting neural network model working on raw audio data, and deploying it on edge devices. This will be achieved by implementing, training, and evaluating a model before porting it to an iOS / Android / R-Pi device. A number of datasets will be explored with the final dataset forming the basis of a system activated via trigger words, known as keywords. Literature and exploratory analysis of the available data and deep learning models will be conducted. The analysis will consider the quality of the data and the accuracy and efficiency of the currently available state of the art deep learning models.

Beyond the primary goals of creating a state of the art neural network model that balances the trade-off between classification accuracy and low computational overheads, but also have such a system execute commands based off speech input from a user. Ideally such commands would operate such that the mobile system can act as a personal safety device which executes a warning noise, light activation or security SMS when the user speaks a keyword, possibly indicating that they are in a dangerous situation. This will be edge device dependant and will pose an interesting final task for the conclusion of the project development.

Chapter 1: Introduction

This project aims to design and implement a number of deep learning models in order to reach state of the art accuracy in the field of audio recognition and more specifically keyword spotting. It aims to do this without creating large models, instead focusing on the low parameter approach. Many varying approaches are currently in existence to achieve this today, motivated by the need to have small models operating on the edge with the ever expanding need for high performance on battery powered devices. Battery power consumption is a major factor in the sphere and so must be one of the top considerations when building such models. What will set this project apart is its ability to create such models and also deploy it to edge for use as a live inference making device. The live inferences will exist with the purpose of creating a useful and scalable 'personal security assistant'. For the purposes of this task the most suitable edge device for porting seems to be the Raspberry-Pi device. Whilst not exactly micro-controllers, edge devices such as Raspberry-Pis are small computational systems that will require some serious memory considerations. This poses the issue of building a model that operates with a low latency to ensure it can run on a small edge device whilst still maintaining a high level of classification accuracy.

Initially, a complete literature analysis will be conducted to consider the most efficient, consistent and accurate neural network architecture. The base challenges here include the use of raw audio data which aims to reduce/completely remove the use of pre-processing which has traditionally been the computationally expensive backbone of KWS systems and also extracting the relevant features from this data to achieve a high classification accuracy. The best architecture that will be used in the final model will be the one which can demonstrate the optimal trade-off between low latency and classification accuracy.

The KWS approach is an interesting one and evolves from the requirement of engaging in natural communication with edge devices/smartphones/home assistants. It is not feasible however to engage in constant speech recognition on battery powered devices as this simply consumes excess power on devices already constrained by battery requirements. Instead, the majority of speech recognition systems engage in passive listening, whereby they are 'woken' by keywords, e.g. the 'Hey Google' or the more recently popular 'Alexa'. These trigger words then allow the device to perform the computationally intensive active listening whereby they can stream audio input directly to the cloud for evaluation. In this project the Keyword approach will be used as a trigger mechanism, having the device the model exists on performing simple but effective actions.

Upon review, the most suitable dataset for this project appears to be the Google Speech Commands dataset [1]. This dataset is used across all of the most recent state of the art models in the KWS scene, allowing for an even basis of comparison across literature. As a result, this project will also use the Google Speech Commands dataset for the training and evaluation of the model. This project will use the second version of the dataset, released in 2019 and containing extra keywords, this will allow extra flexibility in the choice of words which can be used as trigger commands.

Beyond the primary goals of creating a state of the art neural network model that balances the trade-off between classification accuracy and low computational overheads, but also have such a system execute commands based off speech input from a user. Ideally such commands would operate such that the mobile system can act as a personal safety device which executes a warning noise, light activation or security SMS when the user speaks a keyword which indicates that they are in a dangerous situation. This will be edge device dependant and will pose an interesting final task for the conclusion of the project development.

There are a number of factors as to why the personal safety aspect is being considered, with ever increasing lifespans the world has never housed so many elderly people, and with the onset of Covid-19 this cohort of people have never been more isolated and vulnerable. Whilst there is obviously incredibly powerful tools to engage with KWS such as 'Alexa' and 'Siri' it is important that there is direct engagement with solutions tailored toward the most vulnerable in our society. Amidst this backdrop, it is easy to see why the pursuit of an accurate, efficient and ultimately useful KWS tool is desirable.

Chapter 2: Related Work and Ideas

Artificial Neural Networks (ANNs) have been around for nearly 50 years but have only recently regained popularity in a renaissance due largely in part to increased computational power. The rapid advent of Human Computer Interaction (HCI) has also spurred an ever evolving interest in the technologies [2]. Commonly also called Deep Learning (DL), DL algorithms and architectures have arrived at the point of outperforming humans in a number of different cognitive tasks, including speech recognition. [3]. Speech Recognition is not a new field unto itself either, with one of the first papers also being published in the late 1960s [4]. This pioneering paper whilst not using ANNs, used a series of physical feature logic gates to achieve 90% accuracy on spoken letters 0-9. The trend today remains to be continuously improving speech recognition devices. We as consumers will have noticed this trend, especially over the last 10 or so years with the introduction of household personal assistants' in Alexa, Siri, Cortana and Google Home to name but a few. All of these fall under the branch of a continually evolving and exciting research area known as Keyword-Spotting (KWS) or Wakeword Detection.

KWS (as separate from speech recognition) is also not new, having been pioneered in the late 80s/early 90s [5, 6] with Hidden Markov Models (HMMs) until it fell into a period of stagnation. HMMs remain highly competitive in their ability to classify keywords with a high level of accuracy, the inherent issue with HMMs, however, is they are computationally intensive and impractical for deployment to edge devices. Mainly due to the requirement for Viterbi Decoding. [7]. On top of this, approaches like HMM require a lot of handcrafting when arranging the features for the Mel Frequency Cepstral Co-efficients (MFCC) and other pre-processing techniques. Naturally, having something require a lot of human engagement can lead to inefficient allocation of resources. [8]. The introduction of direct classification techniques like SincNet and other filterbank algorithms [8-10] has proven to be a game-changer in the KWS realm. However, the core issue remains as to what neural network architecture is most suitable for the given task at hand, in this case being a KWS system. A suitability analysis of different architectures is discussed in further detail below.

2.1 Neural Networks for KWS

These days there are a number of architectures at the forefront of the KWS scene. These architectures include HMMs, Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Convolutional Recurrent Neural Network (CRNN) - implemented in KWS by Arik et al. [11] Temporal Convolutional Neural Networks [12] and finally, Depthwise-Seperable Convolutional Neural Networks (DSConv) pioneered by Xception [13], used in KWS by Mittermaier [14] and in Image Recognition by MobileNet [15]. In typical circumstances, a KWS system is made up of a feature extraction process which builds a filterbank and then a neural network based classifier, however, Mittermaier, Ravenelli [8, 14] have proven that it is possible to remove the feature extraction process that builds the filterbank and instead perform direct classification on raw audio data using Sinc-Convolutions as pioneered by SincNet [8]. Direct classification has proved to be very useful in ensuring a low memory footprint is achieved which leads to high energy efficiency.

Deep Neural Networks

Deep neural networks (DNN) have become the de-facto standard of what we think of when considering neural networks. DNNs generalise well to almost any task that requires a neural network. Much attention has been given to them largely because they draw particular inspiration from biology and human cognition [16]. The same paper states their inherent value in natural language processing which is why many KWS researchers have opted to use them in their architectures such as Tang et al. [17]. DNNs operate using the standard feed-forward model which comprises of a stack of fully connected layers. The layers aim to learn feature hierarchies using features formed from lower level layers. DNNs provide great flexibility and are often used in conjunction with the HMM paradigm to work on speech recognition tasks. [18]

Convolutional Neural Networks

CNNs have rose to prominence in detection problems, most notably from AlexNet [19] which won the ImageNet challenge from ILSVRC 2012 [20]. It has prominence in particular with image detection problems but also speech due to their ability to extract important features from data in the time and frequency domains and use this as a basis for classification. [21]. These networks are made up of 2 main layers, the convolutional layer, whereby the model is fed with a series of vectors to which the layer applies some form of a linear transformation. This is usually followed by the max-pooling layers which perform local max operations over an input sequence.

Convolutional Recurrent

Convolutional Recurrent is best represented by [11] it is a marriage of both convolutional and recurrent layers in order to try get the best features out of both strategies. In [11] a high accuracy is achieved with 97% but there is still 207k parameters used. This is very good considering a lot of models sometimes require millions of parameters but it is not perfectly ideal either as this many parameters have an effect on energy efficiency, contrast this with [14] who achieve similar accuracy on DSConv with only 63k parameters. RNNs often take the form of Long Short Term Memory networks.

Depthwise Separable Convolutional

DSConv architecture has proven to be the most state of the art out of all the architectures discovered in this project. [3, 13–15]. The main benefits of DSConv can be seen in their ability to produce a highly accurate classification results in a compact form, which makes it the ideal candidate for an energy efficient model. DSConv are made up of 2 layers, the first is the depthwise convolution and the second is the pointwise convolution. The pointwise serves to combine the outputs from the depthwise layer. The outputs of which are then sent to a average pooling function.

As can be seen there is a large variety of possible architectures that can be used, each with their own benefits and drawbacks. One of the key factors in choosing an architecture for this project is how well the given architecture can perform under energy efficiency constraints. It is important to carry out a deeper look into such constraints.

2.2 Energy Efficiency

Whilst low latency is an absolute must across any application running on a battery powered device, it is not as much of an issue for this project which will implement the model using an edge device such as the Raspberry-Pi as it is for Zhang et al [3]. In this paper the authors are dealing with micro-controllers. The premise of micro-controllers is that they are both cheap and energy-efficient processors. They are becoming ever more omnipresent in our everyday life with their usage factors appearing in devices which ranging from home appliances, automobiles and consumer electronics to wearables. Given that a typical micro-controller can only fit tens to hundreds of Kb of memory makes this constraint exceptionally important. In [3] the most efficient architecture given the micro controller constraint turned out to be the DSConv model which achieved 95.4% accuracy whilst also using the least amount of memory out of all the architectures tested.

The latest State of the Art applications and ANNs have shown that it is entirely possible to reduce power consumption without reducing classification accuracy. Mittermaier et al [14] have proven this in their paper which uses no direct pre-processing on the data and instead classification is done directly in raw audio data. They use parameterized Sinc Convolutions (SincNet) as demonstrated in [8]. The SincNet results are sent to a Grouped Depthwise Separable Convolutional Neural Network (GDSconv) and when compared against the previous SOA techniques which include using Mel Frequency Cepstrum Coefficients as a pre-processing strategy they achieve 96.6 percent accuracy with considerably less parameters (62k) than these traditional measures.

Direct Classification is not necessarily a requirement for low latency. In [7] the authors aim to provide a small footprint KWS using DNNs but they do so using traditional feature extraction methods before sending the data through the DNN. The authors prove that it is possible to create a low latency always on model that does not require direct classification using some of the newer technologies that are available.

As is explained in [22–24] one of the most important factors in creating an energy efficient model is the re-use of weights across the network. This evolves from the assumption that on microcontrollers, reading from memory is considered computationally expensive and therefore places drain on battery power. It can also be considered that data movement is draining so when weights can be re-used it significantly reduces both these outstanding issues which often is a requirement in most network architectures.

2.3 Feature Extraction and Filterbanks

One of the most pressing issues and important steps in ensuring accuracy in ANNs is the initial step of feature extraction for which the model will learn its neural pathways. Traditionally, for speech recognition tasks involving HMMs, feature extraction methods that have dominated have been Viterbi Decoding and Mel Frequency Cepstral Coefficients. Recent works [24] [25] [18] have demonstrated that this traditional approach, whilst a good generalised approach to ANNs, is not an optimised solution for KWS and speech recognition tasks. In particular, [25] take aim at MFCC filterbanks which they believe to be redundant in speech recognition as they are designed with a human perceptual understanding of audio processing and that this does not translate well into speech recognition's primary goal of reducing word error rates. They instead believe that the MFCC banks can be simply converted into a layer joined with a deep CNN. This concept is what also drives SincNet [8]. SincNet proposed a novel CNN architecture whereby the first convolutional layer, being fed with raw audio samples, aims to discover more meaningful filters. It is based on learnable sinc functions whereby the layer only learns the high and low cut off frequencies as

opposed to a simple CNN layer which learns all elements of every filter. This offers a customised filterbank that is specifically tuned for the intended application in a compact and highly efficient way. The convolutional function itself is defined as being

$$y[n] = x[n] \times h[n] = \sum x[l] \times h[n - l] \quad (2.1)$$

Where $y[n]$ is the output of the convolutions, $x[n]$ is the audio signal and $h[n]$ is the sinc filters of length L . As will be discovered through the body of this project, the outputs are highly customisable and do not devolve to being task specific. This holds true as despite being developed and applied to Speaker Verification tasks, we successfully apply it to the KWS sphere and discover the flexibility of this novel architecture. Overall, the SincNet approach defines a paradigm to process general time-series signals and can be applied to many fields.

Figure 2.1. Shows us the basic model premise upon which MFCC works, taking the audio signal, breaking it down and performing the operations on it before outputting the CCs. Whereas in Figure 2.2, we can observe the direct classification technique which SincNet employs, moving away from the need to perform filterbanking and instead use the raw waveform itself.

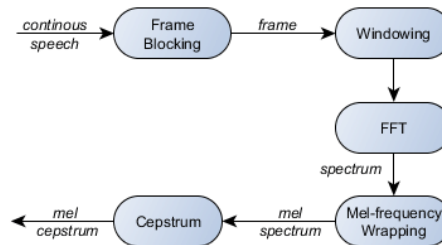


Figure 2.1: MFCC Architecture [26] This block is essentially what the SincNet layer (seen in below) aims to remove. In the frame blocking section, the raw signal is broken into frames to allow for processing. The FFT block converts each frame from the time domain to the frequency domain representation. In the Melfrequency wrapping block the signal is plotted against the Mel-spectrum to allow for feature extraction

It appears to be quite obvious that the energy efficiency of the model can be in some capacity related to the way it is trained and the type of data it is trained on. As such it is important to discuss the reasoning behind the use of and the details of the Google Speech Commands Dataset and why it is suitable for the task at hand in this project.

2.4 Dataset: Google Speech Commands

The Google Speech Commands [1] is one of the most popular open source speech datasets available, due to its single audio snippets nature. In conventional speech recognition training sets the audio streams would have consisted of longer streams i.e. more sentence like structure. This is not efficient or practical for a KWS system. The dataset consists of 105,829 utterances of 35 different words. This particular collection has greatly enhanced the KWS research sector as it allows a single dataset to be used by different researchers which allows for a direct comparison across different methods, architectures and implementations and therefore provides great assistance in bench-marking and reproducibility. The dataset was created with KWS in mind. meaning that most KWS systems require a 'wakeword' which allows them to transition from the passive to the active state of listening. This use of a trigger phrase allows for a much more efficient network as it no longer has a need to send a constant stream of audio to the cloud for processing, which is

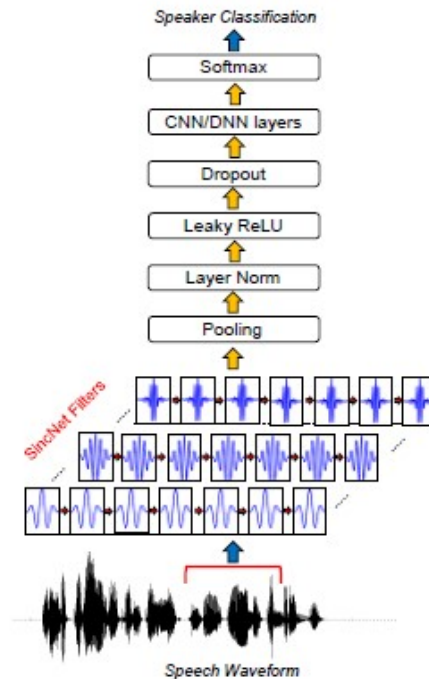


Figure 2.2: Architecture of SincNet.[8] The SincNet layer here performs the convolution with a predefined function, this layer then sends the extracted features into the model for training

both resource intensive and inefficient. As mentioned, the research dataset only has 35 different words which makes it limited in its vocabulary and is something which can limit the end user as to how much can be achieved with the words. As a result, flexibility is required when considering what keywords can be used for what tasks later on in the project.

Baseline Construction and Implementation Ideas

The baseline model will consist of a Convolutional Neural Network learned from a filterbank developed using MFCC. This will serve as a reference point as it has proven to be consistently one of the top performing approaches to KWS since the inception of the technique, this model will also be tuned and adapted to achieve a high level of accuracy in order to make the comparisons non-trivial. The state of the art for this project aims to outperform this approach on the 2 main categories of classification accuracy and computational power usage.

For this project, and given the overall state of the art background research observed here, the approach that will be taken with this project is to adopt a SincNet style filter extraction layer which will then be followed by DSConv layers to classify the keywords provided in the Google Speech Commands Dataset. There currently exists detailed implementations of the SincNet paper on GitHub (provided by the authors). This code base will be interpreted and adapted for the purposes of the feature extraction section of this project. This approach is not novel, nor does it intend to be, it is the most robust approach that the background research has acquired thus far and no doubt it will be a challenge to implement. The approaches outlined here, when combined have only recently started appearing, the 2020 ICASSP conference was one of the first conferences where this style of architecture has appeared. As a result, it provides a very exciting challenge for this project to be at the very forefront of this rapidly evolving industry.

As already mentioned in the specification, the advanced goals for this project include the physical use and implementation of the network on an edge device. There is not much specific literature

which relates to this as it is an extremely project specific goal and will be a novel coding challenge. That being said should this goal be achieved it will make for a very interesting overall project which has true value and use in everyday life, something which is of utmost importance when dealing with a project that relates to human computer interaction.

Training Models - GPU requirements

The training of deep learning models is computationally expensive and more often than not requires powerful machine hardware. Mostly this can be done on local machines provided the machine has access to a reasonable GPU. I do not have access to a GPU directly but luckily there are a number of resources that exist to allow for remote deployment of GPU training. UCD offer a powerful cluster called UCD Sonic which allows for the submission of jobs for completion by the cluster. Another, more viable option is provided by an industry leader in the space with Google's Cloud Platform. The most convenient product in the platform is the Colaboratory notebook resource (known as colab, accessed through research.colab.google.com). This resource allows for an iPython interface with access to Google's own GPU, TPU and CPU resources. This service is provided free of charge under research licensing and enables researchers and students alike to quickly deploy their machine learning models to the cloud for training.

Tensorflow Lite

Tensorflow created Tensorflow Lite (TFLite) in 2019 in order to address the problem of deploying models onto mobile or edge devices. Tensorflow Lite allows for on device inference on such devices which removes the need for server connections or external APIs to perform inference. There is good documentation available on the tensorflow lite website for how to convert and deploy trained models onto embedded Linux systems such as the Raspberry Pi.

Chapter 3: Project Workplan

This project's success can be outlined by developing a state of the art neural network model which is able to outperform the conventional architectures in 2 categories: Classification Accuracy and Low Power Consumption. Ultimately, the end goal is then to port this model to an edge device such as a Raspberry-Pi Device for real-life interactive use. The device should be able to accurately 'wake' upon recognition of the trained keywords. It should not wake arbitrarily as this would violate both the classification and power consumption targets the project sets out to achieve. Figure 3.1 details the breakdown as to how this project will achieve the targets laid out above.

3.1 Accuracy

A method for evaluating the accuracy of the model is to calculate its accuracy scores which is essentially just the total number of correct predictions divided by the number of predictions made. Google's Speech Commands Dataset [1] has training and testing sets which make the evaluating of accuracy quite straightforward. There is no issues with things like bias in this dataset as there is a roughly equal amount of utterances of all the different keywords in the set. A core issue when evaluating accuracy for a KWS system is how the classifier deals with background noise, for this reason the dataset contains snippets with background noise.

3.2 Timeline

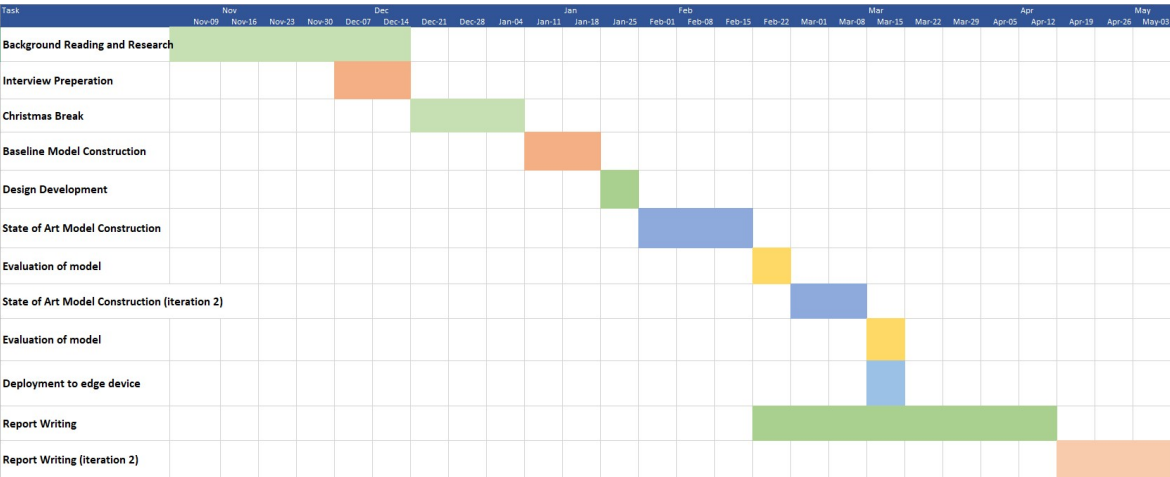


Figure 3.1: Gantt Chart

Background Reading and Research

In order to begin the project it is necessary to delve into the state of the art work currently available in the research world. Due diligence was taken to discover what other leading researchers have

contributed in this field. A large corpus of references was made based on the topics of Keyword Spotting, Neural Networks etc. and from this I drew ideas for my own work.

Interview Preparation

The interviews are scheduled for the 17/12 with the report due the 16/12. It is necessary to have the report wrapped prior to this in order to build the presentation slides and content. Practice interviews as part of the COMP30900 module are scheduled for the 12/12 and therefore it will be necessary to have a draft prepared for this.

Christmas Break

Exams Conclude on the 21/12 meaning there will be approx a 2 week break taken, with work on the project to resume on the 04/01/2021. This gap allows a recharging period and still allows for project work to resume approx 2 weeks before the scheduled UCD trimester on the 18/01.

Baseline Model Construction

Initial exploration has already taken place with MATLAB whereby I have trained an example LSTM model on the Google Speech Commands Dataset. This has allowed me to get a real grasp for what is required, familiarise myself with the process of training a basic KWS system and gives me confidence that I can achieve the goals of the project as specified. A further more in-depth model will be made in Python to serve as a baseline for comparison to the latter SOA model. This is what will take place in the initial stages in the project as is shown in the Gantt chart.

Design Development

It will be important for me to get up to speed with some of the architectures and models I hope to employ on the project e.g. SincNet. This week will allow for all necessary features and packages to be learned and familiarised.

Model Construction and Evaluation (1)

This is the first phase of constructing the state of the art model. A lot of time will be allowed here as this is one of the most important stages of the project. Here the controllable features (hyper-parameters) can be initially set and evaluated to allow for the enhancements to be discovered for implementation in the second iteration. The training of the model will naturally fall under this section and I am allowing for the possibility that training such models can sometimes take quite a while to converge and complete.

Model Construction and Evaluation (2)

Based off the results discovered in the evaluation of the 1st iteration. Incremental improvements can be made on the second iteration, such as requisite adjustments of hyper-parameters. This iteration can be evaluated for improvement until it reaches a satisfactory level of classification ability with the optimal number of parameters being used in the model.

Deployment

This will be the final stage in the project. A shorter window has been allowed here as this is not something that should cause too much issues. To get to this stage on time would be excellent as this will allow the advanced project specifications and enhancements to be completed. The report writing will have been started prior to this and will be the sole task remaining upon completion of deployment to an edge device.

Chapter 4: Summary and Conclusions

In conclusion, there are clear objectives laid out for this project; create a small footprint KWS system using SOA models before evaluating and deploying them to an edge device for active human use. From the findings so far, it is clear that this area is quite well researched and as such many solutions are currently in existence. What separates this project from most of these solutions is the attempt to port and make live inferences on an edge device. Models are normally trained to be as accurate as they can be on a particular dataset and are left at that, which is certainly useful from a research and an advancement of architecture style but it often fails to bring the context of human use into play.

There is without a doubt quite an ambitious scope to this project. However, it is achievable due to the work that has already been done by the current leading researchers in the field as well as the availability of the open source Speech Command's dataset. Google and its researcher's repository of keyword utterances will be the cornerstone of making reliable and benchmarkable models.

The necessity and requirements for this project are indeed plain to see. IoT is a rapidly evolving space and it's use for humanity becoming clearer by the day. The focus of this project to ultimately create a personal safety device aims to merge the state of the art in current neural network research and the everyday user. With people getting older and technology evolving faster it is important to ensure that the fusion of feeling safe and using technology to do so should be at the forefront of what we aim to achieve. From a data and computer science aspect it is an interesting and exciting opportunity to place the power of machine learning and neural network models on full display in their ability to tackle and solve real world problems and future proof ourselves.

Chapter 5: Core Contributions

The first stages of this project involved the exploration of the dataset and to ensure that a firm grounding and knowledge of the dataset existed in order for use in an efficient manner. Due to fact that the dataset consists of raw audio files in .wav format external python libraries were required. Many libraries exist today to allow for the reading of audio files. One such library is Librosa [27]. Librosa is one of the leading Python packages in the audio space. It provides the building blocks necessary to create music and soundwave information retrieval systems. This project utilises Librosa in order to visualise and prepare the audio samples during the data exploration stage. Most people are already familiar with what a common representation of sound-waves looks like. An example of such a wave can be seen here in fig 5.2. Amplitude is hosted on the y-axis with the x-axis containing the time period. All of the .wav files in the dataset are trimmed to 1 second and this can be seen below, throughout the entirety of the project the audio snippets are sampled at a rate of 16000, meaning that there are 16000 samples per seconds of audio. Each keyword contains a number of samples in the range of 2000-4000, indicating a class imbalance might possibly be present that was not present in earlier versions of the dataset, where equal amounts of samples were present for every keyword. This spread of data can be seen in Figure. 5.3. Because we are dealing with raw audio data there is no feasible way to generate additional samples in order to address an imbalance, the only other method for imbalance addressing is to downsample the entire dataset to get the number of samples for every keyword in line with each other. This approach was experimented with but ultimately it was not used as to downsample the dataset to create equal samples would take the total dataset size from 105,000 samples down to approx 60,000 samples. It was decided that the tradeoff was not worth it. It was also noted that current results in literature on this dataset do not attempt to rectify the imbalance and so any comparison of results against literature would make for an uneven comparison as the dataset would have been fundamentally changed.

Frameworks: The project required the use of deep learning frameworks to build and train models on. The two dominant frameworks in the deep learning sphere are Tensorflow and PyTorch. Tensorflow having the high level Keras API [28] as part of its official implementation. Having no familiarity with either framework the initial stages of the project were spent learning and developing using Tensorflow-Keras. This framework was initially chosen due its instant compatibility with Tensorflow lite which, as stated in the background work section, is a pivotal requirement for the porting of models to the edge. However, over the course of the project it was discovered that PyTorch would form an essential part of the construction of the state of the art model, this lay in the fact that there did not exist a sufficient implementation of the work of Ravanelli et al. [10] in the Tensorflow framework. Further investigation found that this would not pose a major problem as there currently exists frameworks such as the Open Neural Network Exchange (ONNX) [29] to allow for the conversion of trained models between different frameworks. Ultimately the conversion process took the form of Figure. 5.1. Consequently we end up having models built through both frameworks.

For training and testing on models, the dataset was split into separate sets. The dataset is arranged and organised into 35 distinct folders with each folder label being assigned to the spoken keyword. In order to create separate sets from the data, it must first be extracted to remove all files from their sub-directories, this is done to enable proper shuffling. Once removed from its sub folder each file gets tagged with its corresponding label. This is done using a Python dictionary. All file locations are stored in a flattened master list which then gets shuffled and split into corresponding training and test sets at a rate of 20%. The training data then gets split further into a training

and validation set at a 10% rate. This is to enable observation of model performance during the training phase.

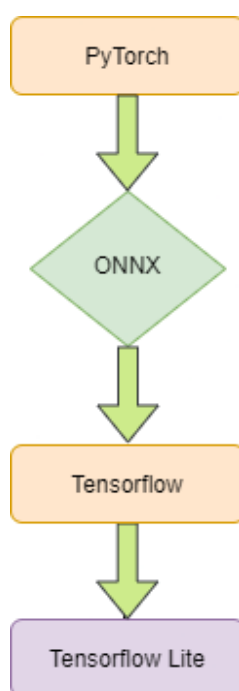


Figure 5.1: Conversion process required for PyTorch to enable use on Raspberry Pi

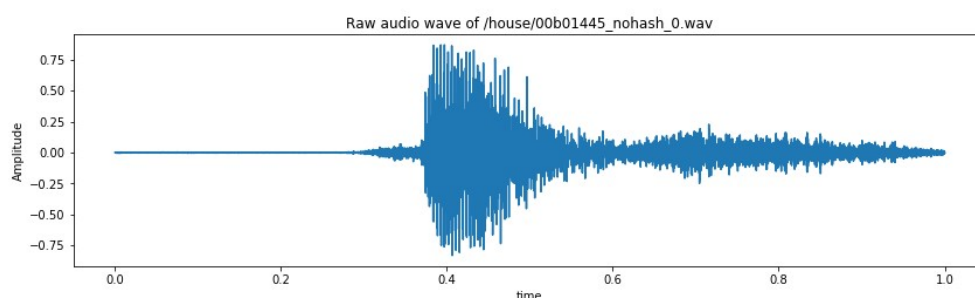


Figure 5.2: Raw Waveform of a speaker uttering the keyword 'house'

5.1 Model Architectures/Breakdown

Baseline

MFCC-CNN

The MFCC model will serve as a useful baseline for comparison against the the state of the art. As mentioned previously, MFCCs have been the backbone of successful KWS systems for quite sometime. The idea of creating MFCC representations is to enable the conversion of audio in time domain into frequency domain via getting the power of each frame of audio. This is based on the human interpretation of sound whereby the cochlea in our ear has more hairs (filters) at lower frequencies and less at higher frequencies. Depending on the location in the cochlea that vibrates (which wobbles small hairs), different nerves fire informing the brain that certain

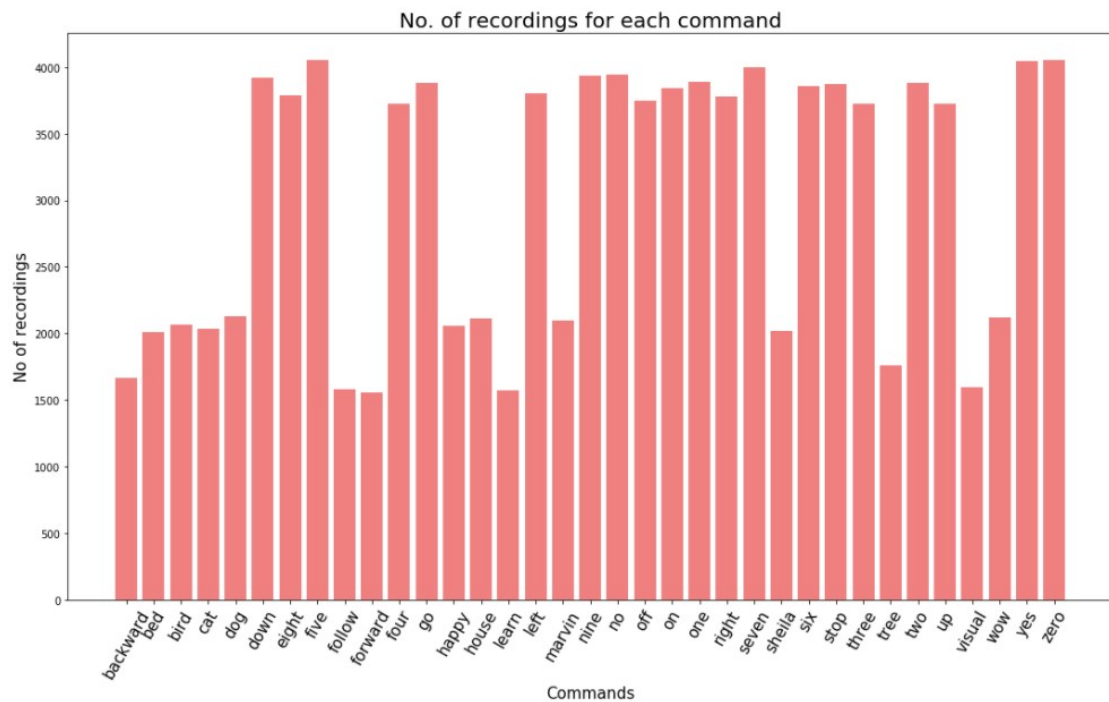


Figure 5.3: Distribution of count of samples of the 35 keywords in the Speech Commands dataset [1]

frequencies are present - this is the estimate within the frames by which the MFCCs attempts to give us [30]. A graphical representation of an MFCC for the keyword 'house' is seen in Figure 5.4. They are a tightly compacted representation of a spectrum of sound and there is essentially a 6 step process for obtaining their values: Framing the signal (splitting the signal into overlapping windows) → A discrete Fourier Transform (DFT) is applied to the frames, this is generally in the form of a Short Time Fourier Transform (STFT) → Once this is completed we can apply a Log Mel spectrogram transformation onto the existing transform (log is used as sound is inherently a logarithmic function) before finally we apply another Fourier transform to result in our MFCCs. There is no need to calculate these manually, as mentioned there are libraries to do this for us, in this case we use a library called `python_speech_features`, this package allows for the dynamic adjustment of parameters in order to customise MFCC shapes. To achieve the desired shapes a function was created, tuning of this function is straightforward and allows different shapes of features to be created. Through the process of trial and error most successful input shape created for use on the MFCC models was 13x99. The model topology that follows the input layer is a 3 layer deep CNN, it can be seen in Figure 5.5 Each Conv2D layer is formed into a block consisting of a MaxPooling and a dropout, dropouts are required to prevent any over-fitting that occurs. Over-fitting is a common problem in deep and machine learning and the process of adding a dropout layer has proven successful in combating this. In this case a dropout rate of 0.25 is added at the end of each block, meaning that after each block 25% of the units will be set to 0. The use of MaxPooling allows us to keep the model small. These blocks are consistent with the exception of the middle block where it can be seen that an extra Conv2D layer has been applied. This layer acts as a 'bottleneck' where we go from 64 to 16 features - allowing the model to summarise information before the final block. All Conv2D layers employ Rectified Linear Unit (ReLU) activations. Two fully connected dense layers are utilised to bring the number of features down to 35 where a softmax activation then determines the most likely keyword. A full summary of the model can be seen in the Appendix section 9.

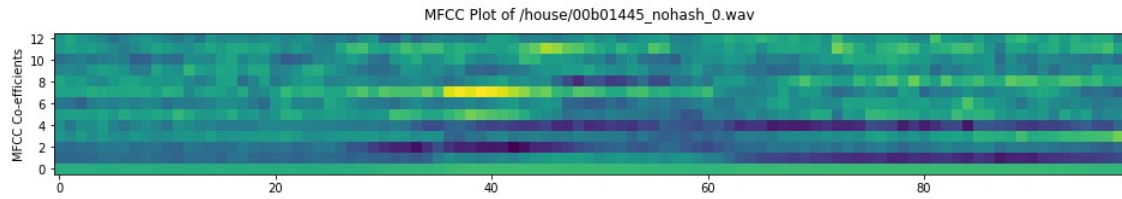


Figure 5.4: Mel Frequency Cepstrum Co-efficient plot of a speaker uttering 'house', The same utterance as can be seen in [5.2](#)

SincConv-DSConv

The idea behind creating a SincConvolution layer is straightforward, as seen in the previous models the notion of speech signals is heavily influenced by the human perception of sound, i.e. how our cochlea interprets sound waves as electrical signals before sending to the brain for interpretation. However, computers are not restricted by a cochlea, so why place such a restriction on them when attempting to learn signals? This is why the first convolutional layer in the SincConv model *learns* the mel bands that are present in the speech signals from the raw audio. We can see the ability of the SincConv layer to produce learnable outputs with the analysis of a spectrogram, Figure [5.6](#) displays the Mel Spectrogram output of the SincConvolution layer, we can directly compare this to a manually calculated Mel Spectrogram output in Figure [5.7](#). Both of these spectrograms represent the same speaker uttering the word 'four' and the same forking style shape is observable in both figures.

The final model topology emulates that of [\[14\]](#) In that the first layer of input is the Sinc-Convolutions developed by [\[10\]](#). The layers can be visually represented as in [2.2](#), however, we only employ the convolution layer. This layer is activated by a custom log layer of the form.

$$\log(\text{abs}(x) + 1)$$

This custom activation has a proven to be successful in other architectures for CNNs trained on raw audio. [\[9\]](#)

This is followed by 5 distinct blocks of Depth-wise Separable Convolutions. The DSConv topology can be seen in more detail in Figure [5.8](#). This topology can be conceptually explained via a number of stages. The first stage involves the splitting of the in channels (this is consistent as 160 in our case) into 'groups', this is controlled in Pytorch by setting the groups as being equal to the number of input channels. The convolution then occurs on a per group basis with the corresponding filter applied. This produces the output tensors where they then get stacked (corresponding to the 3rd bottom stack in Figure [5.8](#)). From here the additional pointwise convolution occurs across the channels with a 1x1 spatial filter.

In [\[14\]](#) the authors implement a form of Grouped Depthwise Separable Convolutions (GDSCConv) in order to further parallelise the model but in this paper we exclusively use DSConv layers. Fig [5.9](#) highlights our resulting block topology, As can be seen the DSConv is performed, activated by ReLU then sent to a BatchNormalisation [\[32\]](#) layer followed by an average pool in order to shrink the model and finally a dropout layer of 0.2 is applied in order to prevent overfitting. A full overview of the model summary can be seen in the Appendix [9.1](#). Note the 2 convolutions represent the depthwise separable combination with the first layer performing the depthwise convolution and the second performing the pointwise convolution. We activate all layers except the final linear layer with ReLU and the linear layer with a softmax. Certain aspects and details of the model architecture proposed in [\[14\]](#) are left blank and therefore require some tuning on our end. Padding is added at each stage of convolution in order to preserve the shape and the output shape of each block is only reduced via the average pooling of 2, this ensures that the second dimension

of each our output i.e. the time dimension gets reduced by a factor of approximately one half, padding notwithstanding. The final pooling layer pools the last 15 dimensions into a single vector to create a 160x1 feature vector which gets flattened and sent to a linear layer. We employ a loss function of sparse categorical cross entropy loss which has proven successful in models pertaining to multi-class classification, it is only recommended for use with the softmax activation function. The Adam optimiser [33] is used as the optimisation function. According to Kingma et al. [33] the algorithm is "computationally efficient, has little memory requirement, invariant to diagonal re-scaling of gradients, and is well suited for problems that are large in terms of data" as such it makes for an excellent solution in our model. We employ the optimizer with a stable learning rate of 0.001 which is commonplace. The model is trained with a batch size of 128 and is trained over 600 such batches at each epoch. Testing of the model found peak accuracy of 94% . Despite its relatively deep 5 layer nature the network remains remarkably small at only 124k parameters (2k more than [14]). This is solely thanks to the use of the DSConv layers.

MelSpectrogram

The MelSpectrogram model is based on the same topology as the SincConv model. It's purpose is to demonstrate any improvements/ worsening of model performance when compared to the SincConv model. As mentioned above, the SincConv model aims to learn the MelSpectrogram using cutoff bands and as such it is worthwhile to train a model which uses a pre-computed spectrogram to form a basis of comparison against. The model utilises the torchaudio MelSpectrogram transform. The parameters of this transform are initialised in order to create a direct replication of the shapes of the SincConv output, this is in order to keep the comparison on the most even possible basis. The resulting parameter list in the transform is as follows:

```
torchaudio.transforms.MelSpectrogram(sample_rate=sample_rate, win_length=101,
hop_length=8, n_mels=40)(waveform)
```

a hop_length of 8ms is chosen in order to get a size of 2000 (sr=16000 per second, divided by hop length of 8ms =2000). n_mels determines the other aspect of our shape. We set this to be 40 as recalling from the SincConv model (see Figure. 9.1 Appendix.) the output shape of SincConv prior to pooling is 40x2000. We therefore create 2 equal spectrograms, the caveat being one is learned via convolutions and the other is pre-computed in the traditional manner. The full output summary of this model can be seen in the Appendix under Figure. 9.3. This model is trained over a batch size of 128, with 600 batches per epoch. The model which achieves the best validation loss after each epoch is the one which gets saved for evaluation on our test, achieving a best accuracy of 83% on the Speech Commands dataset. 7.1

5.2 HyperParameter/Model Tuning

Hyperparameter tuning forms a crucial part of training successful neural network models. Hyperparameters are researcher controlled inputs that are used to control the learning process during model training. The primary hyperparameters used for tuning include the learning rate, the batch size and the number of epochs for which the model is trained over. The learning rate is regarded as the most important hyperparameter used for tuning. Hyperparameter tuning was first performed on the Keras built MFCC-CNN model. Keras-tuner was the external library chosen for this task. The tuner is presented with a list of learning rates and samples all of them in order to calculate the most successful one. A list of [1e-2, 1e-3, 1e-4] was sent to the tuner and the LR with the highest performance was 1e-3 (0.001). This is used in conjunction with the Adam optimiser. The

Adam optimiser is a variation of the traditional Stochastic Gradient Descent (SGD) optimisation algorithm and so a model was also trained using this optimiser, it achieved a lower performance accuracy than the Adam optimiser however and so we stick with Adam. Figure 5.10 displays a plot of training accuracy vs validation accuracy and training vs validation loss for the MFCC-CNN model. The model employed a callback function which stops training at the point in which the model begins to overfit i.e. when training accuracy continues to increase whilst the validation accuracy plateaus or decreases. The patience of this callback is set to 5, meaning that the model will continue to run for 5 epochs after the first signs of deviation are seen and if there is no improvement after 5 epochs then the training is stopped and the best model saved. It is observable in this plot that there is a deviation from training and validation curves at approximately the 15th epoch. For the SincConv model the training and validation losses are present in Figure 5.11. The model is trained over 60 epochs, we employ a learning rate scheduler which engages in a learning rate decay of 0.5 every 10 epochs. The model which achieves the lowest validation loss after each epoch is the one which gets saved for evaluation on our test set.

The SincConv architecture we work with has been extensively hypertuned by the authors in [14]. Therefore we engage with some model experimentation on the architecture itself. Two additional models were trained on the Speech Commands dataset, one of these had a DSConv block removed and one had an additional DSConv block added. The results of this are seen in table 5.1 where we can observe a number of interesting points. The model which had an additional block actually outperforms the original architecture in our trials, achieving a peak accuracy of 95% on the test set. However, this comes at an additional cost of 28k parameters. The model with a DSConv block withheld achieves a slight reduction in accuracy, hitting a peak of 93% on the test set, this does however come with the added bonus of a reduction of 27k parameters.

Table 5.1: Comparison of SincConv model results on the Speech Commands dataset. [1]

Model	Accuracy	Parameters
SincConv-DSConv: Original	94%	124k
SincConv-DSConv: 4 Blocks	93%	97k
SincConv-DSConv: 6 Blocks	95%	152k

The tradeoff between accuracy and parameters becomes more clear here, the model which gets used for further evaluation later in the paper is the model based on the original architecture, that is the one with 5 DSConv blocks.

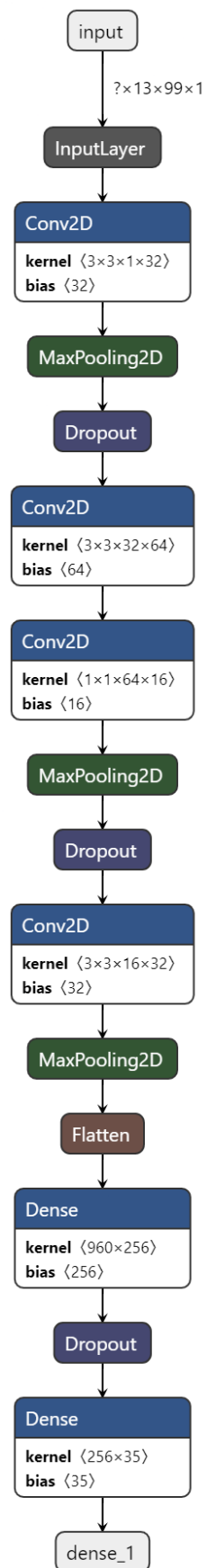


Figure 5.5: Model Architecture employed for the baseline MFCC CNN

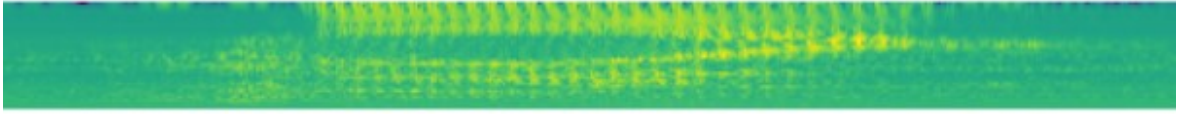


Figure 5.6: Spectrogram output generated by the sinc-convolution layer when the keyword 'four' passed through the layer.

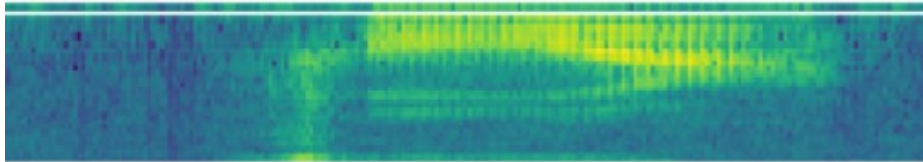


Figure 5.7: Spectrogram output generated by the torchaudio MelSpectrogram package using the keyword 'four'.

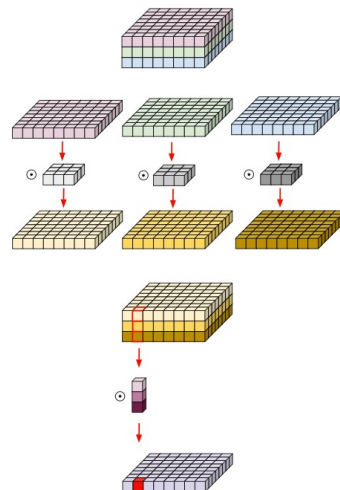


Figure 5.8: A visual representation of Depthwise Separable Convolutions [31], First step is a Depthwise convolution where the number of groups is equal to the number of filters, this creates our depthwise channels, the pointwise convolution then creates a linear combination of the output of the depthwise.

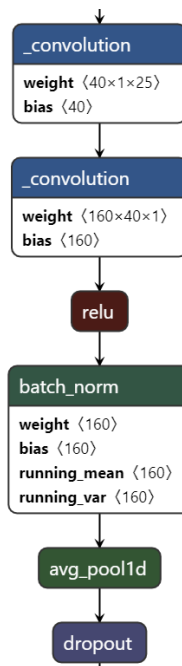


Figure 5.9: The DSConv block, this is repeated 5 times in the model.

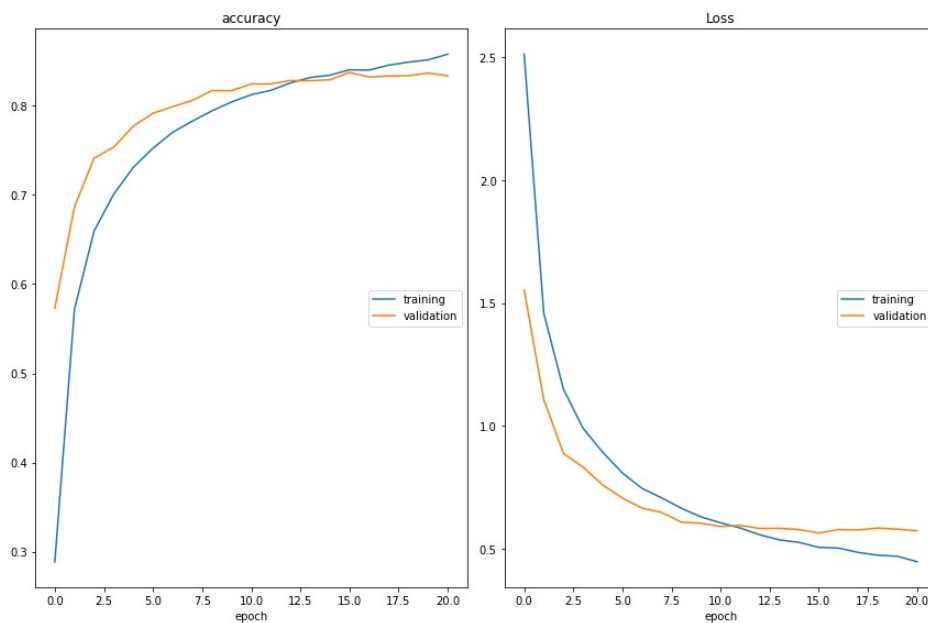


Figure 5.10: Training & Validation plots of accuracy & loss: MFCC-CNN model over 20 epochs.

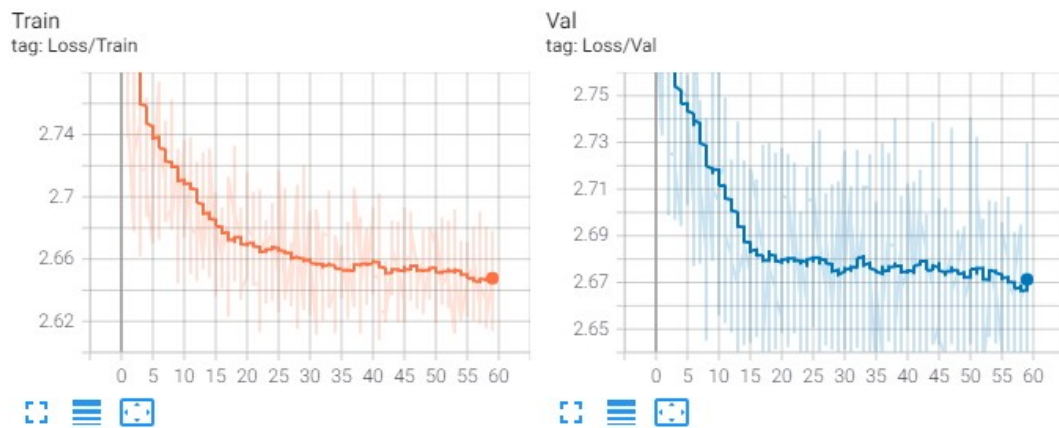


Figure 5.11: Training & Validation plots of loss: SincConv model over 60 epochs.

Chapter 6: Software Engineering/ Deployment

This chapter will focus on the setting up of and deployment of models to the edge device - a Raspberry Pi model 4b. This particular model utilises a BCM2711B0 quad-core ARM processor with 4GB of RAM. The device comes with a number of ready to use ports, including 4 USB A, 2 micro-HDMI, one Ethernet, one 3.5mm audio out jack and one microSD card slot. On it's own the Pi does not have much functionality and for the purposes of this project a number of peripherals were required to make it useable as KWS edge device.

Setting up the Pi: There is a considerable level of work involved in getting the Raspberry Pi device up to speed to match our requirements. The first step in setting up the edge device is the installation of an operating system on the Pi via an SD card. This requires the download an image of the latest version of Raspberry Imager (Details and link found in the Appendix here [9](#)). The Imager provides a useful GUI to ensure the latest version of Raspberry Pi OS gets installed (in this project we use kernel version 5.10.17). The Imager etches the image of the OS onto the SD card for you, alternatively the OS image can be downloaded and etched to the SD using balenaEtcher. In this project we access the pi via SSH, SSH access is denied by default and must be activated prior to installation on the Pi, this is done by creating a blank .txt file titled 'ssh' in the boot drive of the SD card. Once completed we were able to eject the SD card and place it into the SD card slot under the Pi. see Figure [6.2](#). To allow internet connection on the Pi is we solely enable it through Ethernet connection and so we do not configure it to work with WiFi. Internet connection is enabled easily via the connecting of an Ethernet cable from the router to the Pi. To SSH into the Pi it was necessary to gather it's IP address, this information is attainable by logging into the relevant router's login page and viewing devices connected to the router, in this case we had an IP of 192.168.0.220. Using PuTTY for Windows we logged in by entering this IP into the IP box, Figure [6.1](#). From here the pi was configured. VNC viewer was used to access the GUI of our Pi. In our Raspberry Pi ssh terminal VNC access had to be enabled, this achieved by accessing the config properties of the device, via the command `sudo raspi-config`. This opens the settings terminal, where VNC was enabled through Interface Options -> VNC -> Enable. It was also a requirement to update the display resolution to work with our local machine. Display Settings -> choose aspect ratio 16:9, save and exit. Once completed it was then possible to connect fully into the device using VNC.

Once everything was updated we were able to start installing all relevant dependencies. By default python 3 comes installed on the latest version of Rasberry Pi OS. Next most importantly was the installation Tensorflow lite for embedded Linux. Details for installation can be found in the Appendix [9](#). The following commands were used:

```
echo "deb https://packages.cloud.google.com/apt coral-edgetpu-stable
main" | sudo tee /etc/apt/sources.list.d/coral-edgetpu.list
curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo
apt-key add -
sudo apt-get update
sudo apt-get install python3-tflite-runtime
```

which successfully installed the most up to date version of Tflite onto our system. We can then simply use pip3 commands to install the required packages, these packages include numpy,

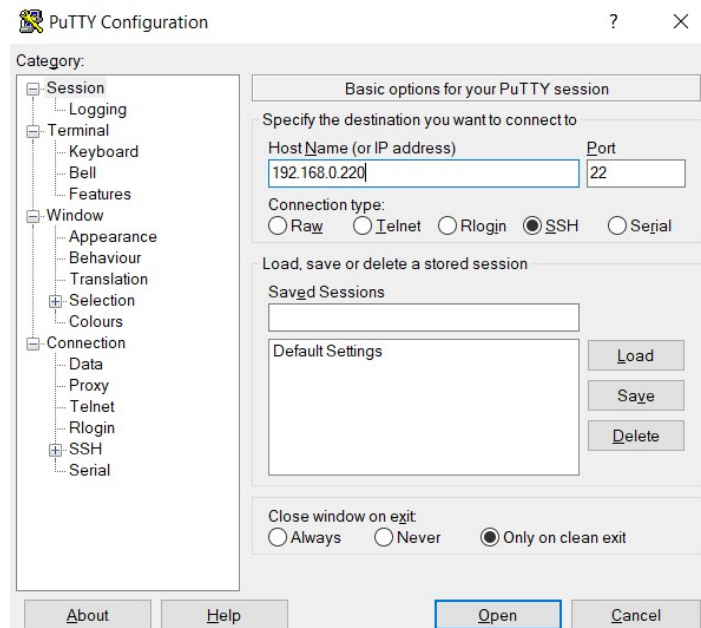


Figure 6.1: PuTTY tool for SSH in Windows connecting to the Raspberry Pi.

`python_speech_features`, `scipy` and `sounddevice`. It is worth noting that some issues were encountered with running `numpy` after installation, such issues were resolved with following `sudo` command: (Appendix 9

```
sudo apt-get install libatlas-base-dev
```

The next phase of set up was ensuring that all peripheral devices ran smoothly and so certain configurations were downloaded and enabled for both the microphone input and the audio out. In order to use the microphone another external package was required. The 'audacity' package was used for this and was installed via:

```
sudo apt-get install audacity
```

The penultimate exaction was the establishment of correct audio re-routing. This was to enable sound to play through the connected speaker system. This was done by again accessing the configuration -> system settings -> audio -> headphones. A final stage of set up on the Pi is creating the Simple Mail Transfer Protocol (SMTP) set up. The relevant downloads are completed via the following install:

```
sudo apt-get install ssmtp
```

Once this is completed we need to configure the SMTP server to accept gmail and our related gmail account. This work is detailed in the Appendix. Once this is completed SMTP is now available for use on the Pi, however a final stage is bypassing Gmail's default security settings, this work is also detailed in the Appendix 9. We are now ready to start porting our files over to the raspberry pi. For this a file transfer protocol (FTP) is required. This project uses FileZilla. Connecting the raspberry pi to FileZilla is a similar process to setting up the ssh service whereby the ssh login details are placed in FileZilla's header bar.

The .wav file is included in the gitlab repository under the folder Raspberry Pi. This needs to be ported into the raspberry pi using the FileZilla method outlined above. The relevant scripts are also ported to the Pi with this approach.

A number of additional external hardware components are also required. This includes the use of an external speaker to be connected to the audio out jack (details in the Appendix). The Pi does

not natively accept audio input and so there are a number of devices which can be used in its stead. The approach this project took was the use of an external USB microphone with a built in audio card, details in Appendix: 9. This was a relatively straightforward approach and requires minimal set up. The set up and device can be seen in Figure. 6.2

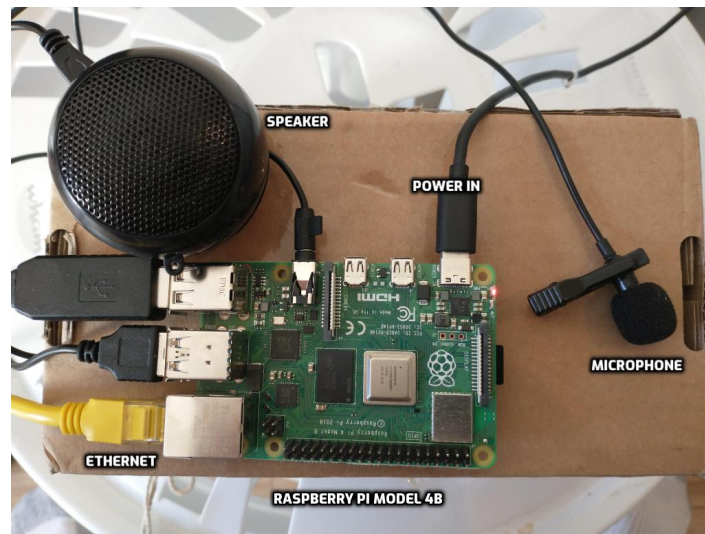


Figure 6.2: The full set up of the Raspberry Pi with peripherals attached to enable live audio streaming, internet connection and audio output.

6.1 Edge Device as a Security Assistant

As stated in the abstract and introduction the project aims to be deploy the models in order to engage in HCI. The project operates 2 primary functions in this regard. Whilst the model is trained to recognise 35 keywords as was intended, it will only utilise 2 of these keywords when performing live inference on the Pi, these keywords are 'house' and 'down'. When spoken, the 'house' command sets off a loud alarm. This is intended as an aide to help deter any home intruders. The second feature keyword is 'down'. This keyword is intended as an aide to the elderly/infirm in the event they suffer a bad fall and cannot reach for help, when spoken this keyword sends an email through the SMTP protocol to a predefined ICE email contact. A number of custom python scripts are created for performing the required tasks. There are 2 main scripts, 'HeyIntruder.py' and 'HeyIntruderMFCC.py'. Both these perform the same tasks with the only caveat being the model used for inference. These scripts perform the listening, inference and performing of actions. The smtp.py script is a basic script which contains the email login details of the account, the message to be sent and the recipient email address. This script gets called by the HeyIntruder scripts.

6.2 Approach

The MFCC-CNN model was the first model ported. The model was constructed natively in tensorflow and so the conversion over to TFLite was straightforward. When performing active listening it is important to engage with sliding windows so that word utterances can be heard in full and will

not be caught between frame captures. As such a sliding window is set up so that recording snippets are captured every 0.5 seconds and those snippets are of 1 second duration. This essentially means that for every 1 second in real time, 2 snippets are sent to the model for inference. In order to make live inferences the feature extraction process is essentially the same as with making inference on the test data, that is the snippets are first transformed using MFCC and then sent to the model for inference. Because the model is a multiclassification model, there is a restriction placed on what level of prediction confidence can trigger an action. In order for an action to be triggered the model must return the keyword as being the only possible word spoken, this is achieved in 2 ways, the first being that the index of the keyword must be the argmax produced by the model output, the second is that the probability that it was in fact this keyword spoken must be above the threshold, in this case the threshold is set to be 0.7, that is the model is 70% confident that this is in fact the keyword being spoken. The tflite SincConv model does not require the extra pre-processing step that the MFCC version goes through, due to the fact we are able to classify on raw audio, instead we simply need to reshape the incoming signal so it fits the dimensions set out on the model during the training phase. Each window is sampled at an initial rate of 48000 as this is the default microphone sampling rate, because all models were trained using a rate of 16000 means that the signal must undergo a decimation in order to be interpreted correctly by the models. Although the microphone settings are configured by default to stream in mono (1 dimension) and not stereo (2 dimensions) we employ a failsafe conversion to squeeze any possible stereo signal into a 1d signal. Once these conversions are completed it is then simply a matter of sending the signal to the compressed model. The process for triggering actions is based on the passing of the 'if' statement requirements:

```
if(prediction == 13 and output_data[prediction] > word_threshold):  
  
    my_sound.play()  
    flag=1
```

Where prediction equal 13 indicates the value of the key 'house' and the model confidence prediction is stored in output_data, so an inequality comparison against the pre-defined word_threshold value is required. The script uses the built in Python pygame library, an intuitive library that allows for the easy playback of audio files in python scripts. In this case 'my_sound' is pygame Sound object which contains the file of the 'Intruder-alert.wav' alarm sound. The flag variable is set to 1 after the sound gets played, this ensures the script stops making further inferences but does not stop the alarm from sounding, the alarm can be turned off manually after it is activated.

Drawbacks

Because of the nature of the dataset we are somewhat restricted in terms of what keywords can be used, the dataset as discussed is 35 keywords and consists of a variety of generally unrelated words e.g. 'sheila', 'zero', 'yes' etc. This applies a restriction on the type of keywords that can be used as trigger words. In order to find a balance between obscurity and usefulness the two keywords that were chosen were 'house' and 'down'. However, these have the possibility of posing problems in the real world as they are not entirely obscure. If used as a functioning security assistant it would not be ideal to have an emergency SOS message being sent without there being an emergency. Possible solutions to this are examined in the Evaluation section where the use of Online Deep Learning is explored. Another solution to this would be engaging with false positive error detection, this can be implemented by engaging with the previous stream of input, i.e. the detector will only activate if there is a pre-defined period of silence between the word being spoken and the intervening period afterwards. This would remove the possibility of the device being activated by a keyword that was innocuously spoken mid-sentence or similar.

Chapter 7: Evaluation

The models can be evaluated through a number of criteria. The most important number when it comes to any model is firstly its accuracy number. That is the percentage of correct predictions in relation to the number of total predictions made by the model. In this project a comparison the accuracy of the baseline MFCC model and MelSpectrogram model is made against the SincConv model. All models are trained on the same dataset splits with a random seed of 42 and they also use the same unseen testing data. More specific to this project is another metric which is model size, Model size is traditionally measured by its total number of parameters. As discussed previously model size directly affects the computational efficiency of a model, therefore achieving high accuracy on smaller models is desirable. Another metric this project uses is power consumption of the model and its related systems on the edge device. To measure this metric an ammeter is placed in between the power source and the Raspberry Pi unit. The unit can measure the uptake in power needed by the Pi when the model runs and performs an inference which is followed by a decisive action (email alert or loud alarm sounds). Comparing the performance of the models across these varying metrics will allow us to develop a clearer picture as to which model is most suitable for making inferences on an edge device.

Accuracy and Size

Table 7.1 highlights the performance of the 3 models across the 2 metrics of accuracy and size. The SincConv-DSCnv model achieves an accuracy of 94% on 124k parameters which is just below the current absolute state of the art of 96.6% with 122k parameters[14]. Despite being below the current state of the art the accuracy shown is right up there and when compared to the baseline of the MFCC-CNN we observe notable improvements in both metrics. We also see some very surprising results when comparing the SincConv-DSCnv to the MelSpec-DSCnv. As mentioned, the only distinction between these models comes from the first layer. The disparity in the accuracies of the 2 models must therefore be attributable to the features that these first layers are generating, this is a solid affirmation of the quality of the sinc-convolution layer of learning directly on raw audio data.

Table 7.1: Comparison of model results on the Speech Commands dataset. [1]

Model	Accuracy	Parameters
SincConv-DSCnv	94%	124k
MelSpec-DSCnv	82%	124k
MFCC-CNN	88%	220k
SincConv-DSCnv as reported in [14]	97%	122k

Power Consumption

As has been identified previously an important metric for evaluating a model on the edge is the power required to run such a model. Once the models are deployed to the edge they can be monitored for their level of power consumption. Once such approach for this is the placement of a digital ammeter between the power source and the device, this set up is shown in Figure. 7.1. The models were measured separately in as controlled an environment as possible. The MFCC-CNN model and the SincConv models are the 2 models that were deployed to the edge, therefore they will form the comparative evaluation. There is some fluctuation of consumption

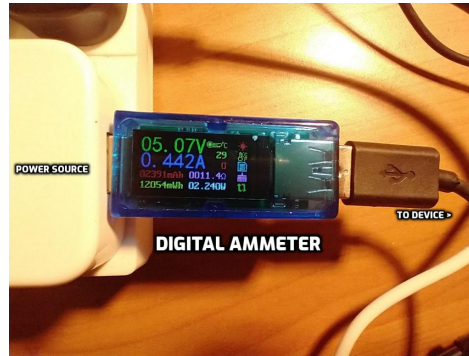


Figure 7.1: Ammeter in use. Measurement of the Raspberry Pi when idle. i.e. it is not performing any inference

over time, therefore the average power consumption over the course of 2 minutes was taken, this was performed by taking a reading every 6 seconds. The results of this experiment are seen in Figure. 7.2.

Current (Amps)	
SincConv Model	MFCC Model
0.84	0.71
0.78	0.77
0.71	0.76
0.7	0.72
0.72	0.78
0.82	0.81
0.74	0.73
0.73	0.76
0.81	0.78
0.73	0.76
0.76	0.8
0.7	0.81
0.76	0.74
0.82	0.72
0.7	0.77
0.63	0.7
0.71	0.82
0.85	0.72
0.71	0.77
0.85	0.83
0.7535	0.763

Figure 7.2: Results of current consumption test. 20 results were taken for each model over the course of 2 minute period. Average over the 20 results is seen in bottom row.

The SincConv model performs marginally better with an average of 0.01 less amperes in comparison to the MFCC-CNN model. There is a certain level of idle current that the device consumes, this idle current can also fluctuate, although less severely. Therefore due to this fluctuation the idle current is not subtracted from the figures seen in the experimental results, instead the idle current is included in the results seen. However, it can be noted that the idle current is measured at approx 0.433 amps. If this is taken at face value we can conclude that power consumption based entirely on model requirements would equal 0.3205 and 0.33 for the SincConv and MFCC models respectively. We can perform rough calculations based on these results with some assumptions, assuming that this consumption is universal, that is the models will consume an equivalent amount of power no matter the edge device (i.e. iPhone, Android etc.). And taking manufacturer battery specs at face value then we can calculate how long a model running on such a device would take to fully drain the battery. Working on the assumption that the Samsung Galaxy S20 (a market leader in battery performance) has a battery size of 4.5Ah and making the entirely naive assumption that

the only thing consuming power on the device would be the model we get the following results Figure. 7.3: It can be observed that over an extended period the SincConv model allows for an extra 0.5hrs or 30 minutes of additional battery life when evaluating on the above assumptions.

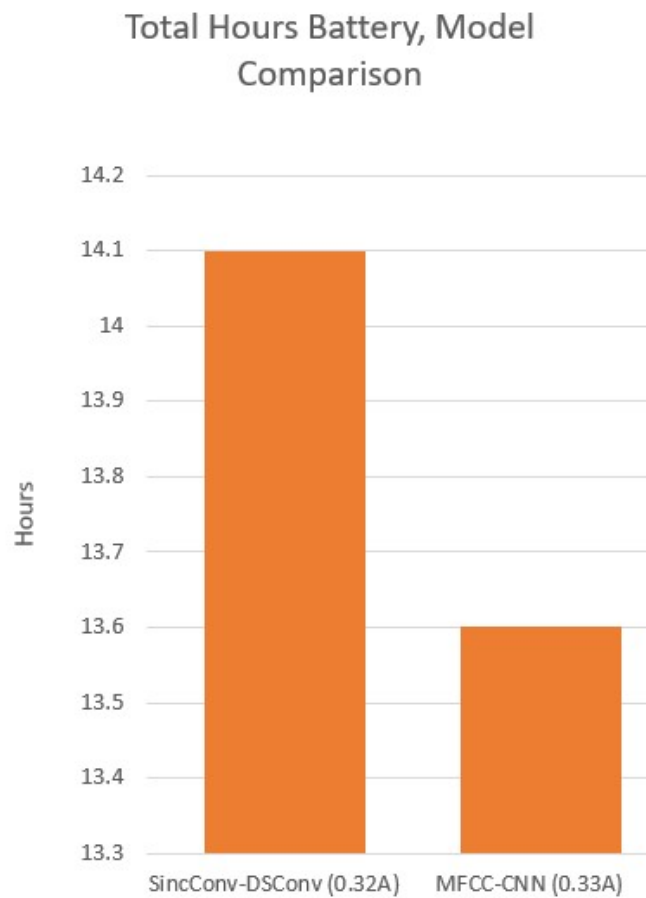


Figure 7.3: Battery life (hrs) of edge device with battery capacity of 4500 mAh (4.5Ah) based solely on respective model consumption rates.

Having established that the SincConv model outperforms the baselines through a variety of metrics, the next focus of evaluation is on the flexibility and robustness of the SincConv model. In this section there is some direct evaluation tasks performed on the model. The first task of evaluation is through the addition of background noise. The ability of a keyword spotting device to be robust in the presence of noise is an important one as sound is an omnipresent feature in modern life. A final method of evaluation focuses on the structure of the model itself and it's ability to remain flexible. This is explored through the naive use of Online Deep Learning (ODL) where the model is used a feature extractor in order to add additional unseen keywords into the prediction mix.

Addition of noise

In order to be robust and useful in a real world environment, models should be adept to spotting keywords in the presence of background noise. In this project we train the models solely on the original data and do not engage with any additive noise in the training set, this is in order to keep the model reproducible and in line with the project specifications. However, we can still perform tests and observe how the trained model performs on a test set which *does* have noise added to it. The noise we add is artificially created white noise generated through a Gaussian distribution using NumPy [34]. The level of noise to be added is determined by the Signal-to-Noise-Ratio

(SNR), for this project an SNR in the range of $(-20, 20)$ is employed where -20 means there is total additive white noise, that is the amplitude level of white noise in the audio clip supercedes that to the original amplitude of the original clip. An SNR value of 0 means there is equilibrium between the amount of background noise and the amount of original audio. A representation of this addition is seen in Figure. 7.4. In this figure we utilise an SNR value of -20 , indicating that there is quite a substantial amount of background noise. This is clear in the plot as the original wave is completely enveloped by the white noise.

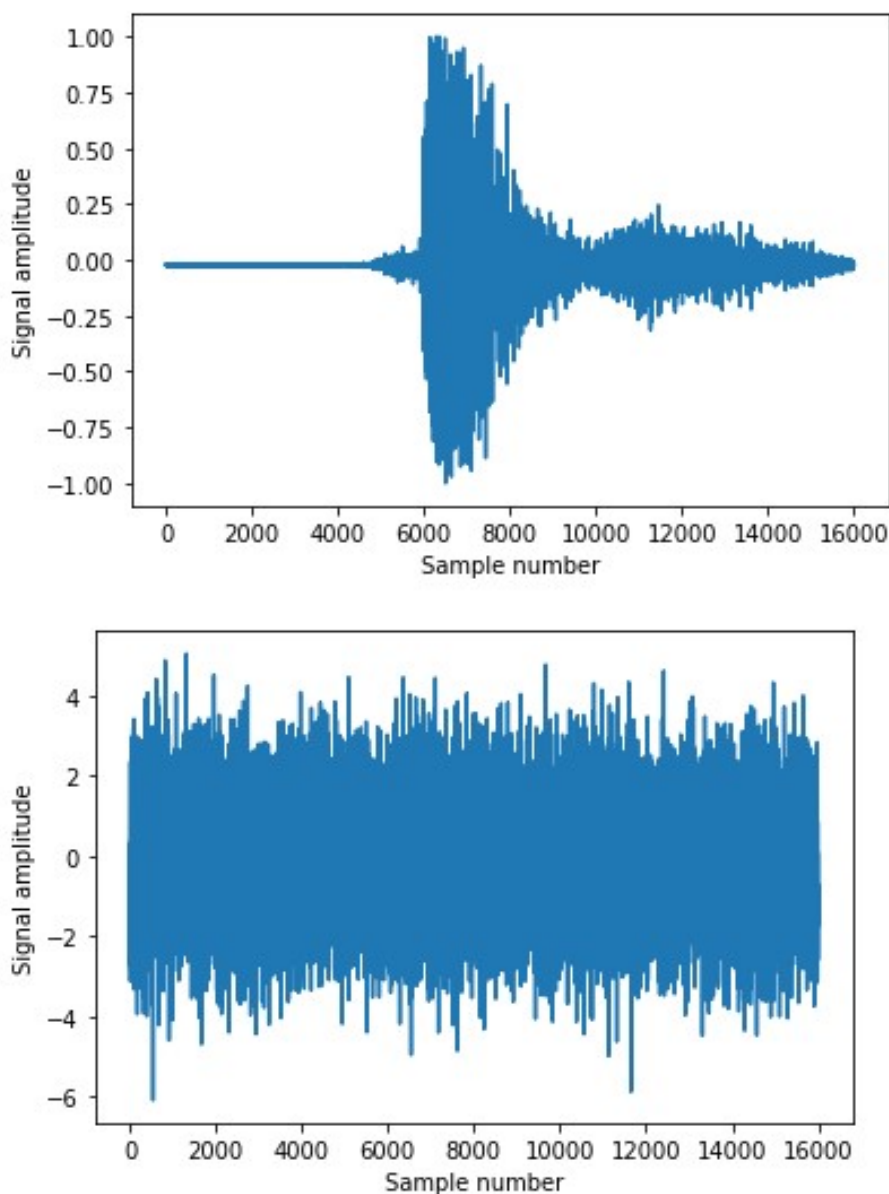


Figure 7.4: 'House' before and after white noise addition. SNR value of -20 , indicating that there is 20x more noise present in comparison to the level of audio

The noise is added via an 'on-demand' basis, that is the noise is only applied once the model signal is extracted from the audio files, this is the most feasible way to perform this step as we wish to experiment with many different levels of noise addition. The results of this experiment can be seen in Figure. 7.5. The model stays very robust up until an SNR value of ~ 10 where accuracy remains at approx. 90%, after this point increasing SNR by even small amounts results in a sharp decline of accuracy. As was observed from 7.4 however, this is a large augmentation of the wave so the model is still engaging in high levels of accuracy despite the conditions, although it certainly loses its state of the art tag at these levels.

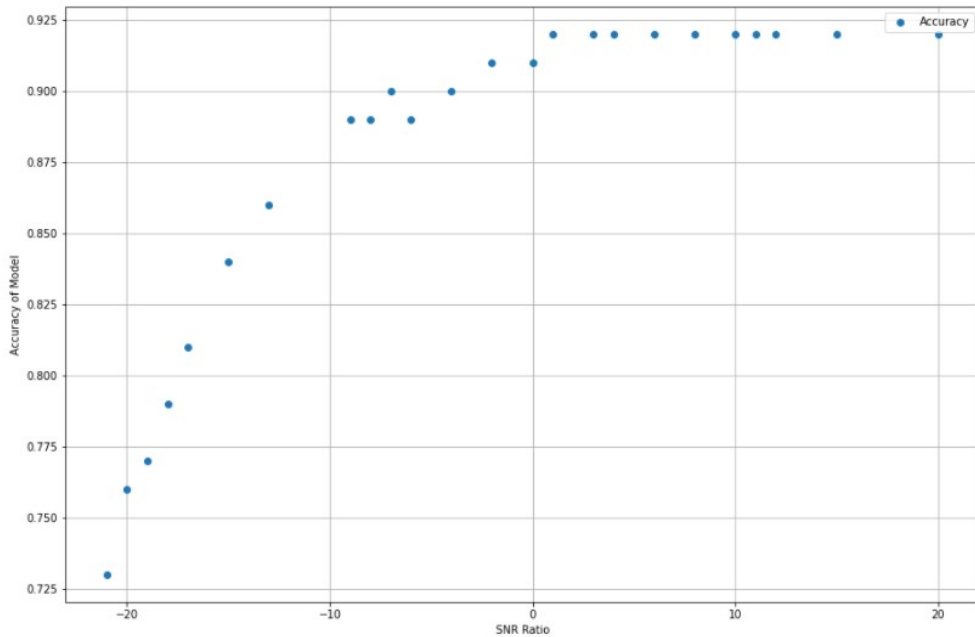


Figure 7.5: Plot of Accuracy vs Signal to Noise Ratio on SincConv Model

Online Learning/Out of Core Learning

As has been shown, neural networks trained in the traditional manner via back-propagation and batch learning produce robust results. In the SincConv model trained the final fully connected flattened layer gets sent to one further linear layer to down-sample the predictions into the 35 class labels which gets activated by a softmax. While this is highly effective it is also limited, the network is learning features but is then restricting these features to being exclusively used for labelling a predefined number of keywords. There exists a field within the deep learning sphere called Online Deep Learning (ODL). Online learning is an exciting approach to enable trained models to learn and classify on previously unseen inputs and there currently exists a number of useful libraries for large scale implementations such as LIBOL [35]. The ODL approach allows for more scalable real world scenarios where data arrives sequentially in a streamed format. It is inefficient and computationally infeasible to relearn models at every iteration of new data. One naive implementation approach is to 'chop' the legs off an already learned model, i.e. remove its activation function and send inputs through the network to receive an N dimensional feature vector of 'learned' outputs and then perform a binary classification using linear regression modelling on the outputs of the chopped model. In this case N is 160 as this is the output of the final layer prior to activation. This layer can be seen in the summary of the model in the Appendix. The idea therefore is that if the model is robust and produces generic features then *any* keyword that passes through the chopped model can produce learnable features. This is implemented in this project in order to analyse if the feature vectors produced by the model are in fact generic and robust. There were a number of approaches that could have been taken to analyse this, the fully trained SincConv model described in section 5.1 could be used to generate features and additional custom audio could have been provided by me. However, there is a large body of work in creating a custom corpus of sounds and so an alternative approach was taken to demonstrate the ODL implementation. The approach taken was to retrain the model, this time only using 34 out of the 35 keywords, reserving the extra keyword as our unseen input for the linear regression step later on. With the trained model we can now generate a matrix of feature vectors extracted from audio samples. We generate a subset of 4000 random samples from our 34 existing classes and a further range of (100, 4000) samples of our unseen keyword, in this case the keyword is 'zero'. Because we only care about the classification of this new unseen word we devolve the labels into binary classification, assigning a value of 0 if it is not our new keyword and a value of 1 if it is.

With a matrix of features and a vector of labels we can fit a simple linear regression. This is done using the Python package Sci-kit Learn [36]. We instantiate a LinearRegression model from the linear_models package. This is an Ordinary Least Squares regression, meaning that the model aims to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. The fitting of such is as straightforward as:

```
reg= LinearRegression(normalize=True)
reg.fit(X, y)
```

Where X is the our matrix of features and y is consists of the ground truth labels. The accuracy of the linear regression model was observed through incremental addition of the unseen samples into the model. A plot of accuracy vs no. of new samples can be observed in Figure 7.6. As can be seen

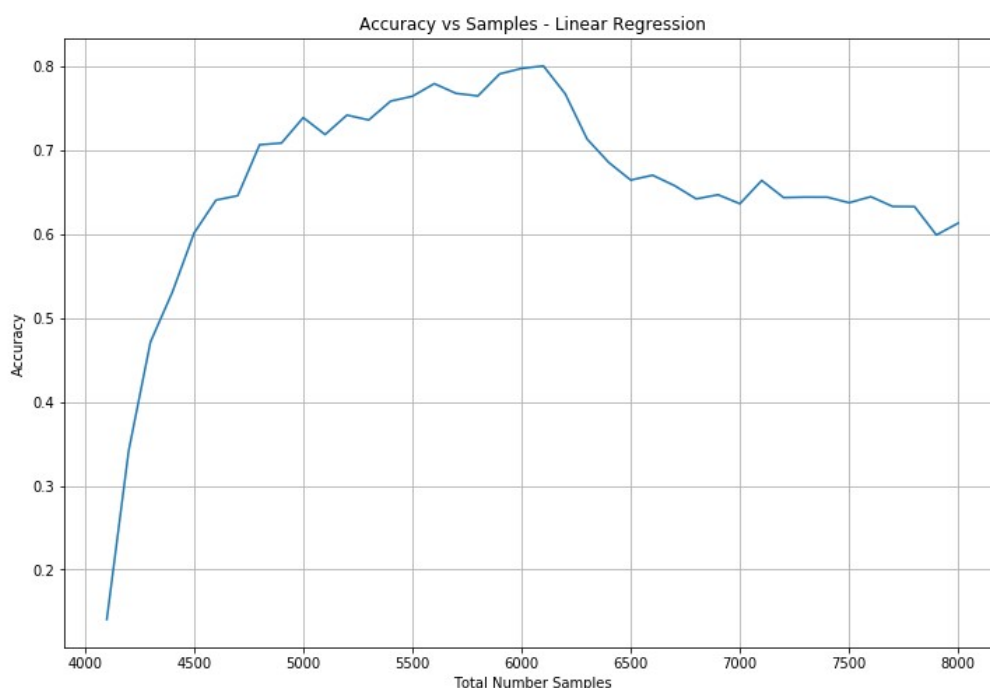


Figure 7.6: Plot of Accuracy vs Number Samples on adjusted SincConv Model (34 classes). Number of labels 0 is fixed at 4000 with the increment coming in the form of new unseen 'zero' data. Peak accuracy is reached with 2100 zero samples

the accuracy reaches a peak of 80.0% at the addition of 2100 'zero' samples, after which point accuracy drops and plateaus, possibly indicating that overfitting occurs after this point. Thus, what this shows is our 'chopped model' is producing generic features that are not dataset specific and can be utilised for extra audio data. Once again highlighting the robustness and usefulness of this particular architecture across a variety of metrics and implementations.

Chapter 8: Future Work

There is ample capacity for further exploration beyond the scope of this particular project. Whilst many avenues are indeed examined, many more remain open for further development and research. A highly interesting and exciting facet of the project was encountered towards its terminus with the use of ODL, developing custom corpora for more personalised keyword additions would be a compelling task and would enable a highly elegant solution on the edge. There is also an enhanced scope available for the establishment of a more user-friendly interface on the edge device, this could be achieved via the use of smartphone porting, for example Apple have have an excellent iOS based facility for model porting known as Core ML. It would also be possible to create a GUI for the Raspberry Pi itself, although it must be noted that the usage of the Pi often requires pre-requisite knowledge of technical experience in the first place. The models are trained on the entire corpus of the dataset, meaning that when performing live streaming the model will always classify a word as being spoken regardless as to whether is has been spoken or not, this can be improved by training the model with a 'silence' or even an 'unknown' tag, that is that there is a fallback for the model to allow for more efficient KWS when performing live inference. Finally, further research into the use of raw audio processing possibly more specific to the KWS realm could be developed with more focus on the short snippet nature of keywords. Creating low parameter, highly accurate models is an exciting and challenging task and no doubt there will be many more developments within this space in the years to come.

Chapter 9: Conclusion

In conclusion the project ends with overall success, state of the art models were trained and deployed to the edge. The low parameter-high accuracy approach works well with the use of Depthwise separable convolutions which have proved their incredible ability to reduce parameters with no compromise on accuracy. The comparison against regular 2D convolutions shows a halving of parameters without sacrificing accuracy. The benefits of using raw audio processed through the sinc-convolutions shows huge promise and its use is generic, having been initially developed for Speaker Verification/Recognition it has proven that it is transferable to virtually any audio tasks. Our state of the art model does not achieve the exact same high results seen in [14], we instead notice a 2% drop in accuracy but performance still remains at the cutting edge of what is currently available. The model shows extreme promise when ported to the edge, slight reductions in power consumption were noted when compared to the baseline. Despite never being exposed to noise during the training phase we notice a remarkable ability of the model to remain accurate despite large volumes of noise. This was anecdotally tested on the edge when making live inferences, playing loud background noise whilst uttering a keyword still generates a successful match. The advanced goals of the project were achieved with the successful porting of models to the edge where live inferences from a continuous stream of audio can be accurately predicted. We complete the work by having 2 of our 35 keywords perform a task in relation to personal security with:

- The activation of a loud alarm in the presence of the keyword 'house'. 9
- The sending of an emergency email alert to a predefined email address in the presence of the keyword 'down'. 9

It has been shown that these keywords are very flexible and can in fact be anything the user wishes both with the use of ODL and also with the 33 other remaining keywords being available for use.

Acknowledgements

This work was completed by the author Cian Ferriter with the aide of Prof. Guénolé Silvestre.

Appendix

Model Summaries

```
=====
Layer (type:depth-idx)                   Output Shape                  Param #
=====
--Sequential: 1-1                        [1, 40, 1000]                --
  --SincConv: 2-1                        [1, 40, 2000]                80
  --LogAbs: 2-2                          [1, 40, 2000]                --
  --BatchNorm1d: 2-3                     [1, 40, 2000]                80
  --AvgPool1d: 2-4                       [1, 40, 1000]                --
--Sequential: 1-2                        [1, 160, 250]                --
  --Conv1d: 2-5                          [1, 40, 500]                 1,040
  --Conv1d: 2-6                          [1, 160, 500]                6,560
  --ReLU: 2-7                            [1, 160, 500]                --
  --BatchNorm1d: 2-8                     [1, 160, 500]                320
  --AvgPool1d: 2-9                       [1, 160, 250]                --
  --Dropout: 2-10                        [1, 160, 250]                --
--Sequential: 1-3                        [1, 160, 125]                --
  --Conv1d: 2-11                         [1, 160, 250]                1,600
  --Conv1d: 2-12                         [1, 160, 250]                25,760
  --ReLU: 2-13                           [1, 160, 250]                --
  --BatchNorm1d: 2-14                    [1, 160, 250]                320
  --AvgPool1d: 2-15                      [1, 160, 125]                --
  --Dropout: 2-16                        [1, 160, 125]                --
--Sequential: 1-4                        [1, 160, 62]                 --
  --Conv1d: 2-17                         [1, 160, 125]                1,600
  --Conv1d: 2-18                         [1, 160, 125]                25,760
  --ReLU: 2-19                           [1, 160, 125]                --
  --BatchNorm1d: 2-20                    [1, 160, 125]                320
  --AvgPool1d: 2-21                      [1, 160, 62]                 --
  --Dropout: 2-22                        [1, 160, 62]                 --
--Sequential: 1-5                        [1, 160, 31]                 --
  --Conv1d: 2-23                         [1, 160, 62]                 1,600
  --Conv1d: 2-24                         [1, 160, 62]                 25,760
  --ReLU: 2-25                           [1, 160, 62]                 --
  --BatchNorm1d: 2-26                    [1, 160, 62]                 320
  --AvgPool1d: 2-27                      [1, 160, 31]                 --
  --Dropout: 2-28                        [1, 160, 31]                 --
--Sequential: 1-6                        [1, 160, 15]                 --
  --Conv1d: 2-29                         [1, 160, 31]                 1,600
  --Conv1d: 2-30                         [1, 160, 31]                 25,760
  --ReLU: 2-31                           [1, 160, 31]                 --
  --BatchNorm1d: 2-32                    [1, 160, 31]                 320
  --AvgPool1d: 2-33                      [1, 160, 15]                 --
  --Dropout: 2-34                        [1, 160, 15]                 --
--AvgPool1d: 1-7                        [1, 160, 1]                  --
--Flatten: 1-8                          [1, 160]                     --
--Linear: 1-9                            [1, 35]                      5,635
--Softmax: 1-10                         [1, 35]                      --
=====
Total params: 124,435
Trainable params: 124,435
Non-trainable params: 0
Total mult-adds (M): 34.98
=====
Input size (MB): 0.06
Forward/backward pass size (MB): 4.52
Params size (MB): 0.50
Estimated Total Size (MB): 5.08
=====
```

Figure 9.1: Full Model Summary of SincConvolutions Input with DSConv layers. Generated using torchinfo summary package v. 0.0.9

Raspberry Pi set ups

Installing Tensorflow Lite : <https://www.tensorflow.org/lite/guide/python?hl=nb>

Debugging NumPy installation : <https://numpy.org/devdocs/user/troubleshooting-importerror.html>

Raspberry Pi Model 4B purchased here: <https://thepihut.com/collections/raspberry-pi/products/raspberry-pi-4-model-b>

```

=====
Layer (type:depth-idx)                   Output Shape                   Param #
=====
BatchNorm1d: 1-1                         [2, 40, 2001]                 80
Sequential: 1-2                           [2, 160, 494]                 --
├─Conv1d: 2-1                             [2, 40, 989]                  1,040
├─Conv1d: 2-2                             [2, 160, 989]                 6,560
├─ReLU: 2-3                               [2, 160, 989]                 --
├─BatchNorm1d: 2-4                       [2, 160, 989]                 320
├─AvgPool1d: 2-5                         [2, 160, 494]                 --
├─Dropout: 2-6                           [2, 160, 494]                 --
Sequential: 1-3                           [2, 160, 243]                 --
├─Conv1d: 2-7                             [2, 160, 243]                 1,600
├─Conv1d: 2-8                             [2, 160, 243]                 25,760
├─ReLU: 2-9                               [2, 160, 243]                 --
├─BatchNorm1d: 2-10                     [2, 160, 243]                 320
├─AvgPool1d: 2-11                       [2, 160, 243]                 --
├─Dropout: 2-12                         [2, 160, 243]                 --
Sequential: 1-4                           [2, 160, 118]                 --
├─Conv1d: 2-13                           [2, 160, 118]                 1,600
├─Conv1d: 2-14                           [2, 160, 118]                 25,760
├─ReLU: 2-15                               [2, 160, 118]                 --
├─BatchNorm1d: 2-16                     [2, 160, 118]                 320
├─AvgPool1d: 2-17                       [2, 160, 118]                 --
├─Dropout: 2-18                         [2, 160, 118]                 --
Sequential: 1-5                           [2, 160, 55]                  --
├─Conv1d: 2-19                           [2, 160, 55]                  1,600
├─Conv1d: 2-20                           [2, 160, 55]                  25,760
├─ReLU: 2-21                               [2, 160, 55]                  --
├─BatchNorm1d: 2-22                     [2, 160, 55]                  320
├─AvgPool1d: 2-23                       [2, 160, 55]                  --
├─Dropout: 2-24                         [2, 160, 55]                  --
Sequential: 1-6                           [2, 160, 24]                  --
├─Conv1d: 2-25                           [2, 160, 24]                  1,600
├─Conv1d: 2-26                           [2, 160, 24]                  25,760
├─ReLU: 2-27                               [2, 160, 24]                  --
├─BatchNorm1d: 2-28                     [2, 160, 24]                  320
├─AvgPool1d: 2-29                       [2, 160, 24]                  --
├─Dropout: 2-30                         [2, 160, 24]                  --
AvgPool1d: 1-7                           [2, 160, 1]                   --
Flatten: 1-8                             [2, 160]                       --
Linear: 1-9                               [2, 35]                         5,635
Softmax: 1-10                            [2, 35]                         --
=====
Total params: 124,355
Trainable params: 124,355
Non-trainable params: 0
Total mult-adds (M): 75.68
=====
Input size (MB): 0.64
Forward/backward pass size (MB): 10.36
Params size (MB): 0.50
Estimated Total Size (MB): 11.49
=====

```

Figure 9.2: Full Model Summary of MelSpectrogram Model with DSConv layers. Generated using torchinfo summary package v. 0.0.9

External USB Microphone for use on the Raspberry Pi purchased here: https://www.amazon.co.uk/gp/product/B076M4HXFH/ref=ppx_yo_dt_b_asin_title_o04_s00?ie=UTF8&psc=1

Ammeter for power consumption measurement purchased here: https://www.amazon.co.uk/gp/product/B07FNJS7RV/ref=ppx_yo_dt_b_asin_title_o01_s00?ie=UTF8&psc=1

External Speaker 3.5mm audio jack purchased here: https://www.amazon.co.uk/gp/product/B07FCL67YH/ref=ppx_yo_dt_b_asin_title_o04_s00?ie=UTF8&psc=1

Email Configuration

Once package outlined is downloaded we utilise the nano editor to configure the protocol with our credentials, accessed with the command:

```
sudo nano /etc/ssmtp/ssmtp.conf
```

We then place our credentials in the file as seen in Figure. 9.4

Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 11, 97, 32)	320
max_pooling2d (MaxPooling2D)	(None, 11, 48, 32)	0
dropout (Dropout)	(None, 11, 48, 32)	0
conv2d_1 (Conv2D)	(None, 9, 46, 64)	18496
conv2d_2 (Conv2D)	(None, 9, 46, 16)	1040
max_pooling2d_1 (MaxPooling2D)	(None, 9, 23, 16)	0
dropout_1 (Dropout)	(None, 9, 23, 16)	0
conv2d_3 (Conv2D)	(None, 7, 21, 32)	4640
max_pooling2d_2 (MaxPooling2D)	(None, 3, 10, 32)	0
flatten (Flatten)	(None, 960)	0
dense (Dense)	(None, 256)	246016
dropout_2 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 35)	8995
Total params: 279,507		
Trainable params: 279,507		
Non-trainable params: 0		

Figure 9.3: Full Model Summary of MFCC Model with Conv2D layers. Generated using keras model summary. Tensorflow v. 2.4.0

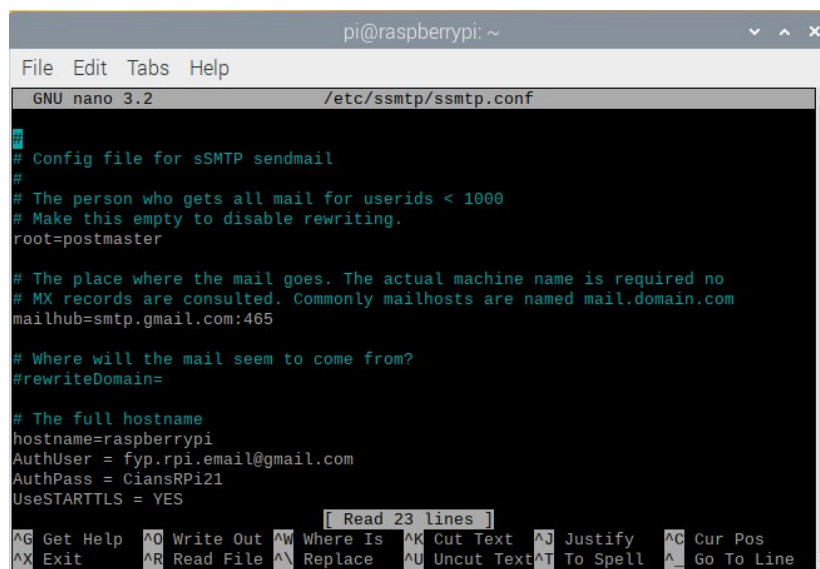
Demos

VIDEO DEMONSTRATION OF KEYWORD 'HOUSE':

<https://youtu.be/So7RcKI2u4Y>

VIDEO DEMONSTRATION OF KEYWORD 'DOWN':

<https://youtu.be/JUDyWr4n1as>



The screenshot shows a terminal window titled 'pi@raspberrypi: ~' with a nano editor open to the file '/etc/ssmtp/ssmtp.conf'. The editor's status bar at the top indicates 'GNU nano 3.2' and the file path. The configuration file content is as follows:

```
# Config file for sSMTP sendmail
#
# The person who gets all mail for userids < 1000
# Make this empty to disable rewriting.
root=postmaster

# The place where the mail goes. The actual machine name is required no
# MX records are consulted. Commonly mailhosts are named mail.domain.com
mailhub=smtp.gmail.com:465

# Where will the mail seem to come from?
#rewriteDomain=

# The full hostname
hostname=raspberrypi
AuthUser = fyp.rpi.email@gmail.com
AuthPass = CiansRPi21
UseSTARTTLS = YES
```

The bottom of the screen displays a status bar with the text '[Read 23 lines]' and a row of keyboard shortcuts: ^G Get Help, ^O Write Out, ^W Where Is, ^K Cut Text, ^J Justify, ^C Cur Pos, ^X Exit, ^R Read File, ^\ Replace, ^U Uncut Text, ^T To Spell, and ^_ Go To Line.

Figure 9.4: configuration requirements for SMTP authentication

Bibliography

1. Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *arXiv*. arXiv: [1804.03209](#) (2018).
2. Lecun, Y., Bengio, Y. & Hinton, G. *Deep learning* 2015.
3. Zhang, Y., Suda, N., Lai, L. & Chandra, V. Hello edge: Keyword spotting on microcontrollers. *arXiv*, 1–14. arXiv: [1711.07128](#) (2017).
4. C. Teacher, H. Kellett & Focht, L. Experimental Limited Vocabulary Speech Recognizer, 3–6 (1967).
5. Rohlicek, J. R., Russell, W., Roukos, S. & Gish, H. Continuous hidden Markov modeling for speaker-independent word spotting. - *In Proceedings ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing, 23-26 May, Scotland 1*, 627–630. ISSN: 07367791 (1989).
6. Rose, R. C. & Paul, D. B. A hidden Markov model based keyword recognition system in - *In Proceedings ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing, 3-6 April, New Mexico* (1990).
7. Chen, G., Parada, C. & Heigold, G. Small-Footprint Keyword Spotting Using Deep Neural Networks. *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 4-9 May, Florence*, 4087–4091 (2014).
8. Ravanelli, M. & Bengio, Y. Speaker Recognition from Raw Waveform with SincNet. - *In Proceedings IEEE Spoken Language Technology Workshop, 18-21 December, Athens*, 1021–1028. arXiv: [1808.00158](#) (2019).
9. Zeghidour, N., Usunier, N., Synnaeve, G., Collobert, R. & Dupoux, E. End-to-end speech recognition from the raw waveform. *In Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 2-6 September, India*, 781–785. ISSN: 19909772. arXiv: [1806.07098](#) (2018).
10. Ravanelli, M. & Bengio, Y. Interpretable Convolutional Filters with SincNet. *NIPS 2018 Interpretability and Robustness for Audio, Speech and Language Workshop, 2018*. arXiv: [1811.09725](#). <http://arxiv.org/abs/1811.09725> (2018).
11. Arik, S. O. et al. Convolutional recurrent neural networks for small-footprint keyword spotting in *In Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH, 20-24 August, Stockholm* (2017). arXiv: [1703.05390](#).
12. Choi, S. et al. Temporal convolution for real-time keyword spotting on mobile devices. *In Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH 15-19 September, Austria*, 3372–3376. ISSN: 19909772. arXiv: [1904.03814](#) (2019).
13. Chollet, F. Xception: Deep learning with depthwise separable convolutions in *In Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, 21-26 July, Honolulu* (2017). ISBN: 9781538604571. arXiv: [1610.02357](#).
14. Mittermaier, S., Kurzinger, L., Waschneck, B. & Rigoll, G. Small-Footprint Keyword Spotting on Raw Audio Data with Sinc-Convolutions in -*In Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing, 4-8 May* (2020). ISBN: 9781509066315. arXiv: [1911.02086](#).
15. Howard, A. G. et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv: [1704.04861](#). <http://arxiv.org/abs/1704.04861> (Apr. 2017).

-
16. Glorot, X. & Bengio, Y. *Understanding the difficulty of training deep feedforward neural networks* in *Journal of Machine Learning Research* **9** (2010), 249–256.
 17. Tang, R. & Lin, J. Deep residual learning for small-footprint keyword spotting. - In *Proceedings IEEE International Conference on Acoustics, Speech and Signal Processing* **22-27 April, Seoul**, 5484–5488. ISSN: 15206149. arXiv: [1710.10361](https://arxiv.org/abs/1710.10361) (2018).
 18. Hiroshi Seki; Kazumasa Yamamoto; Seiichi Nakagawa. A Deep Neural Network Integrated With Filterbank Learning For Speech Recognition. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* 5-9 March, New Orleans, 5480–5484 (2017).
 19. Krizhevsky, A., Sutskever, I. & Hinton, G. E. *ImageNet classification with deep convolutional neural networks* in *Advances in Neural Information Processing Systems (NIPS)* (2012). ISBN: 9781627480031.
 20. Russakovsky, O. et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*. ISSN: 15731405. arXiv: [1409.0575](https://arxiv.org/abs/1409.0575) (2015).
 21. Palaz, D., Magimai-Doss, M. & Collobert, R. *Analysis of CNN-based speech recognition system using raw speech as input* in *In Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH* **6-10 September, Dresden** (2015), 11–15.
 22. Sze, V., Chen, Y., Yang, T. & Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *In Proceedings of the IEEE, Dec 2017* **105**, 2295–2329 (2017).
 23. Chen, Y. H., Emer, J. & Sze, V. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. - In *Proceedings 43rd International Symposium on Computer Architecture, 18-22 June, Seoul*, 367–379 (2016).
 24. Shah Nawaz, M., Plebani, E., Guaneri, I., Pau, D. & Marcon, M. Studying the effects of feature extraction settings on the accuracy and memory requirements of neural networks for keyword spotting. *IEEE International Conference on Consumer Electronics* **2-5 September, Berlin**. ISSN: 21666822 (2018).
 25. Sainath, T. N., Kingsbury, B., Mohamed, A. R. & Ramabhadran, B. *Learning filter banks within a deep neural network framework* in - In *Proceedings IEEE Workshop on Automatic Speech Recognition and Understanding, 8-12 December, Czech Republic* (2013). ISBN: 9781479927562.
 26. Caranica, A., Cucu, H., Buzo, A. & Burileanu, C. On the design of an automatic speech recognition system for Romanian language. *Control Engineering and Applied Informatics* **18**, 65–76. ISSN: 14548658 (2016).
 27. McFee, B. et al. *librosa: Audio and Music Signal Analysis in Python* in (Jan. 2015), 18–24.
 28. Chollet, F. Keras <https://github.com/fchollet/keras>. 2015.
 29. Bai, J., Lu, F., Zhang, K., et al. ONNX: Open Neural Network Exchange <https://github.com/onnx/onnx>. 2019.
 30. Davis, S. B. & Mermelstein, P. Comparison of Parametric Representations for Monosyllabic Word Recognition in Continuously Spoken Sentences. *Transactions on Acoustics, Speech, and Signal Processing* **28**, 357–366 (1980).
 31. Bendersky, E. *Depthwise Seperable Convolutions For Machine Learning* <https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/#id2>. (accessed: 06.06.21).
 32. Ioffe, S. & Szegedy, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* in *Proceedings of the 32nd International Conference on Machine Learning* (eds Bach, F. & Blei, D.) **37** (PMLR, Lille, France, July 2015), 448–456. <http://proceedings.mlr.press/v37/ioffe15.html>.
 33. Kingma, D. P. & Ba, J. L. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations 2015 - Conference Track Proceedings*, 1–15. arXiv: [1412.6980](https://arxiv.org/abs/1412.6980) (2015).

-
34. Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585**, 357–362. <https://doi.org/10.1038/s41586-020-2649-2> (Sept. 2020).
 35. Hoi, S. C., Wang, J. & Zhao, P. LIBOL: A library for online learning algorithms. *Journal of Machine Learning Research* **15**, 495–499. ISSN: 15337928 (2014).
 36. Pedregosa, F. *et al.* Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* **12** (Jan. 2012).